

**VALIDATION OF SCALABLE SOFTWARE-DEFINED NETWORK
SIMULATIONS USING SIMULATION-BASED ROUTING APPLICATIONS**

A Dissertation
Presented to
The Academic Faculty

By

Jared S. Ivey

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2017

Copyright © Jared S. Ivey 2017

**VALIDATION OF SCALABLE SOFTWARE-DEFINED NETWORK
SIMULATIONS USING SIMULATION-BASED ROUTING APPLICATIONS**

Approved by:

Dr. George F. Riley, Advisor
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Henry L. Owen III
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Yorai Wardi
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Richard M. Fujimoto
School of Computational Science
and Engineering
Georgia Institute of Technology

Dr. Russell J. Clark
School of Computer Science
Georgia Institute of Technology

Date Approved: April 26, 2017

When eating an elephant, take one bite at a time.

Gen. Creighton W. Abrams

To my wife Sloane who endured the highs and lows alongside me, never wavering in her support and being the light on even the cloudiest of my days.

To my father David whose work ethic has forever left a lasting impression on me.

To my mother Charla whose education in life has formed me into the adult I have become.

To my sister Devin whose bright spirit has always been able to keep me smiling.

ACKNOWLEDGEMENTS

I must begin by thanking my advisor, Dr. George Riley, whose guidance and encouragement provided an environment where I could excel. As an advisor, he steered me toward exciting and fruitful research endeavors. As a teacher, he imparted countless pieces of knowledge and wisdom. As a friend, he made life around the Ph.D. experience fun, always available for a run or post-conference beer. I am a better person and researcher for knowing him. While thanking Dr. Riley, I must also thank Jacqueline Trappier for introducing me to him. The fortunate encounter that initiated my Ph.D. career also provided a friend in academic advising who made the formal steps of the Ph.D process smooth for me. I would also like to thank Dr. Brian Swenson for everything he continues to do for me as a mentor and friend. He has always been available for support and encouragement throughout my Ph.D. studies. I appreciate his friendship and look forward to staying in touch with him for years to come. Thanking Beverly Scheerer is not nearly enough to reciprocate for all of the things she has done for me. She has acted as my “Georgia Tech mom,” making sure that all of the details behind my daily Ph.D. life run smoothly. Not only has she done all of the paperwork for reimbursements or supplies but also been there to talk about anything and everything to make me feel at home at Georgia Tech. I truly appreciate all of the conversations we have had. I must also thank the Department of Defense and my colleagues at Robins Air Force Base. They had the faith in me to provide this opportunity to earn my Ph.D. in a financially and professionally comfortable manner. Without this support, I could not have even begun this journey, and I thank them for their endorsement. I would like to thank Peggy Dale as well, who helped me understand that I should never sell myself short and gave me the idea that I could accomplish something as initially daunting as earning a Ph.D. As this document is dedicated to them, I also thank my family, who throughout the years have helped me strive to be a better person through their encouragement and example. They are my strength, and I love them more than anything else in this world.

TABLE OF CONTENTS

Acknowledgments	v
List of Tables	ix
List of Figures	x
Chapter 1: Introduction and Background	1
1.1 Contributions	1
1.2 Dissertation Organization	3
Chapter 2: Origin and History of the Problem	4
2.1 Software-Defined Networking	4
2.1.1 The OpenFlow Protocol	4
2.1.2 Controller Libraries	7
2.1.3 OpenvSwitch	9
2.2 Direct Code Execution	10
2.3 SDN Simulation/Emulation Tools	12
2.4 GENI	16
2.5 Routing in Simulation	18
2.6 General-Purpose Computation on GPUs	19

2.6.1	GPUs for Networking and SDN	20
 Chapter 3: Enabling Network Simulation to Support Realistic and Portable Software-Defined Networking Capabilities		
3.1	Extending DCE Beyond C/C++	22
3.1.1	Programming Languages	22
3.1.2	Experiments	27
3.2	Enabling Simulation of GPU Network Applications	35
3.2.1	Native Integration	36
3.2.2	Virtualized Integration	36
3.2.3	Experiments	38
 Chapter 4: Analysis of Existing SDN Simulators and Emulators		
4.1	SDN Simulation with ns-3 and DCE	43
4.2	Experiments	47
 Chapter 5: Deploying Simulation Routing as SDN Applications		
5.1	SDN-Based NIX Vector Routing	60
5.2	GPU-Based Routing with Parallel Floyd-Warshall	64
5.3	Introducing Validated Performance Modeling of SDN Simulation	67
 Chapter 6: Experimental Setup		
6.1	Network Topologies	70
6.1.1	Linear Network	71
6.1.2	Campus Network	71

6.2	Traffic Generation and Observed Variables	75
Chapter 7: Results and Discussion		80
7.1	SDN-Based NIX Vector Routing	80
7.1.1	Linear Network	81
7.1.2	Campus Network	87
7.2	GPU-Based Routing with Parallel Floyd-Warshall	97
7.3	Validated Performance Modeling	101
Chapter 8: Conclusions and Future Work		118
Appendix A: Controller Processing Time Data		121
References		135
Vita		136

LIST OF TABLES

2.1	Packet match fields for OpenFlow 1.0	6
2.2	OXM TLV header fields	7
7.1	Simple sampling results of modeling validation experiments	113
7.2	Feature-based results of modeling validation experiments	114
7.3	Time-based results of modeling validation experiments	115
7.4	Sample-based results of modeling validation experiments	116
7.5	Statistical similarity results of modeling validation experiments	117

LIST OF FIGURES

3.1	Single-node memory usage	28
3.2	Single-node wallclock execution time	28
3.3	Simple dumbbell topology	31
3.4	Simple dumbbell topology memory usage	31
3.5	Simple dumbbell topology wallclock execution time	32
3.6	Ring topology	33
3.7	Ring topology memory usage	34
3.8	Ring topology wallclock execution time	34
3.9	Designs for enabling CUDA support for DCE natively	36
3.10	Design for enabling CUDA support for DCE with gVirtuS	37
3.11	Single-node wallclock execution time	39
3.12	Single-node CPU memory usage	39
3.13	Pairs network topology	40
3.14	Pair wallclock execution time	41
3.15	Pairs CPU memory usage	42
3.16	Pairs GPU memory usage	42
4.1	Overall architecture of OFSwitch13 connecting to a DCE-enabled node. . .	45

4.2	Structure of a DCE-enabled node as a controller and an <code>SdnSwitch</code> object	46
4.3	Linear network topology	48
4.4	Linear network real-time performance	49
4.5	Linear network resource usage	49
4.6	Linear network packet loss	50
4.7	Simplified campus network topology	51
4.8	Ring of campus networks	52
4.9	Campus network real-time performance	52
4.10	Campus network resource usage	53
4.11	Campus network packet loss	53
4.12	Data center network topology	55
4.13	Data center network real-time performance	56
4.14	Datacenter network resource usage	57
4.15	Datacenter network packet loss	57
6.1	Linear network in GENI	72
6.2	Nicol NMS challenge topology	73
6.3	Campus network in GENI	74
7.1	Floodlight RTT (1st Ping) - Linear Network	82
7.2	Floodlight RTT (2nd Ping) - Linear Network	83
7.3	Floodlight Throughput - Linear Network	83
7.4	Ryu RTT (1st Ping) - Linear Network	85
7.5	Ryu RTT (2nd Ping) - Linear Network	85

7.6	Ryu Throughput - Linear Network	86
7.7	Floodlight RTT from Subnet 1 (1st Ping) - Campus Network	88
7.8	Floodlight RTT from Subnet 2 (1st Ping) - Campus Network	89
7.9	Floodlight RTT from Subnet 3 (1st Ping) - Campus Network	89
7.10	Floodlight RTT from Subnet 1 (2nd Ping)- Campus Network	90
7.11	Floodlight RTT from Subnet 2 (2nd Ping)- Campus Network	90
7.12	Floodlight RTT from Subnet 3 (2nd Ping)- Campus Network	91
7.13	Floodlight Throughput - Campus Network	91
7.14	Ryu RTT from Subnet 1 (1st Ping) - Campus Network	93
7.15	Ryu RTT from Subnet 2 (1st Ping) - Campus Network	93
7.16	Ryu RTT from Subnet 3 (1st Ping) - Campus Network	94
7.17	Ryu RTT from Subnet 1 (2nd Ping)- Campus Network	95
7.18	Ryu RTT from Subnet 2 (2nd Ping)- Campus Network	95
7.19	Ryu RTT from Subnet 3 (2nd Ping)- Campus Network	96
7.20	Ryu Throughput - Campus Network	96
7.21	Algorithm Processing Time for Ring of Campus Networks	98
7.22	Algorithm Processing Time for Ring of Campus Networks (Closer View)	99
7.23	Path Construction Processing Time for Ring of Campus Networks	99
7.24	Total Process Completion Time for Ring of Campus Networks	100
7.25	Total Process Completion Time for Ring of Campus Networks (Closer View)	100
7.26	Number of Primitive Python Instructions vs. Wallclock Execution Time	102
7.27	Histogram of Python Controller Processing Times	102
7.28	Histogram for gamma fitting	106

7.29 Histogram for log-normal fitting	106
7.30 Histogram for Weibull fitting	107
7.31 Histogram for gamma fitting with antithetic sampling	107
7.32 Histogram for log-normal fitting with antithetic sampling	108
7.33 Histogram for Weibull fitting with antithetic sampling	108
7.34 Histogram for bound Weibull fitting with antithetic sampling	110
7.35 Remaining histogram for bound Weibull fitting with antithetic sampling . .	110
A.1 Total simulated data for gamma fitting	122
A.2 Subset of simulated data for gamma fitting	122
A.3 Total simulated data for log-normal fitting	123
A.4 Subset of simulated data for log-normal fitting	123
A.5 Total simulated data for Weibull fitting	124
A.6 Subset of simulated data for Weibull fitting	124
A.7 Total simulated data for gamma fitting with antithetic sampling	125
A.8 Subset of simulated data for gamma fitting with antithetic sampling	125
A.9 Total simulated data for log-normal fitting with antithetic sampling	126
A.10 Subset of simulated data for log-normal fitting with antithetic sampling . .	126
A.11 Total simulated data for Weibull fitting with antithetic sampling	127
A.12 Subset of simulated data for Weibull fitting with antithetic sampling	127
A.13 Total simulated data for bound Weibull fitting with antithetic sampling . . .	128
A.14 Subset of simulated data for bound Weibull fitting with antithetic sampling .	128

SUMMARY

With farther reaching applications being developed in the realm of software-defined networking (SDN), simulation can justify the feasibility of deploying initial SDN capabilities in a network or assist with troubleshooting and testing existing SDN deployments as a part of maintenance or expansion. This work describes an SDN simulation framework that supports realistic and portable SDN capabilities. Direct Code Execution (DCE) in the network simulator ns-3 is extended to allow the execution of network programs written in Python and Java. Support for CUDA libraries in DCE is provided, permitting the simulation of portable GPU-based network applications. An SDN simulation framework in ns-3 and DCE is designed allowing scalable, portable simulation of SDN controller applications written for the Python-based libraries POX and Ryu supporting OpenFlow 1.0 and 1.3.

Similarly to simulation, SDN provides an environment where an entire topology is controlled collectively. The mechanisms that are used to manage routing decisions in simulation can be leveraged for use in SDN. Dynamic, on-demand packet routing in SDN is described that exploits currently existing headers and OpenFlow rules to provide a routing solution influenced by Nix vectors. The use of a parallelized version of the Floyd-Warshall algorithm is studied in the context of SDN as well using the massively parallel processing capability of GPUs. With this effort, route generation for scalable SDN topologies is accomplished in less time than with sequential graph algorithms.

The final part of this work aims to provide representative performance profiles that introduce appropriate latencies and other behaviors into the SDN simulation framework. Using multiple Ryu applications, scalable network topologies are tested using both the hardware testbed GENI and network simulations. Controller processing time is gathered and evaluated with the goal of working toward statistically similar results in both environments. A model is designed and evaluated for an adequate representation of instruction processing time distributions in an SDN controller operating in simulation.

CHAPTER 1

INTRODUCTION AND BACKGROUND

Software-defined networking (SDN) originated as a means to separate and logically centralize communication network control from its forwarding plane. This paradigm enables flexibility by allowing networks to be programmable. In its early stages, SDN provided a means for virtualizing network space so networking research could occur and coincide seamlessly with regular traffic on physical campus networks[1]. As SDN grows in both academia and industry, its capabilities expand beyond these initial goals with more use cases elucidating its potential. With farther reaching applications being developed, it can prove beneficial to employ modeling and simulation efforts toward initial developmental testing of these capabilities. Simulation can justify the feasibility of deploying initial SDN capabilities in a network or assist with troubleshooting and testing existing SDN deployments as a part of maintenance or expansion. For initial deployment, simulation of SDN can minimize initial costs of these kinds of exploratory efforts since network hardware is not immediately required. Risk associated with introducing updates to an existing SDN network can be reduced as many issues can be identified and resolved within simulation prior to actual deployment.

1.1 Contributions

The objective of this work is to develop new SDN-based routing applications that borrow concepts from network simulation and deploy these applications to profile their real-world network performance in order to design validated SDN component models. Previous research developing and evaluating an initial SDN simulation framework is presented. The new SDN controller applications that have been developed will then be discussed. The primary contributions of this work are:

- NIX-MPLS flow rule installation behavior for dynamic, on-demand network routing for SDN has been developed borrowing from the originally proposed NIX vector construct. Linear and campus network topologies have been constructed in both GENI and the network simulator ns-3 to examine and compare the performance of controller applications deploying the NIX-MPLS behavior against applications using typical SDN flow rule installation. Experiments are conducted to gather round trip times (RTT) and network throughput under various levels of traffic flow using the *ping* and *iperf* commands. By deploying NIX-MPLS flow rule installation in certain SDN controller libraries, switches can be instructed in a quicker, more succinct manner that, based on the observed results, provides an improved network throughput.
- Path determination utilizing the massive parallelism of GPUs has also been studied, demonstrating improved processing time when using the Floyd-Warshall algorithm. Furthermore, the parallelized version of the Floyd-Warshall algorithm using the GPU provides faster processing times when calculating all paths in an SDN to simpler single-path computation using sequential versions of BFS and UCS. Experiments are conducted on a ring of a variable number of campus networks within ns-3. The controller processing times in terms of algorithm computation, path construction, and general completion are collected and analyzed.
- A model has been designed and evaluated for an adequate representation of instruction processing time distributions in an SDN controller operating in simulation. The approach and pathway to achieving this model has been described, ultimately deeming a shifted and bound Weibull distribution with antithetic sampling as an appropriate approximation. This distribution has been introduced into the task scheduling mechanisms of DCE to influence a random variable stream responsible for providing time values to the event scheduler. These values permit the network simulation to advance simulation time in a way that resembles realistic controller processing times.

1.2 Dissertation Organization

The remaining components of this dissertation are organized into the following chapters. Chapter 2 describes the background topics on which this work has been based, specifically SDN, simulation and emulation in this field, direct code execution as it relates to the network simulator ns-3, simulation routing mechanisms, and the use of GPUs in networks. In Chapter 3, research enabling network simulation is described as it relates to supporting realistic and portable SDN capabilities. A comparison analysis is discussed in Chapter 4 that studies the performance of the SDN simulation framework employed in this work against a number of existing SDN simulators and emulators. Chapter 5 explains the concepts behind the key contributions of this work while Chapter 6 describes the experimental setup used to examine these contributions. Chapter 7 displays the results of the performed experiments and discusses them in terms of their implications on the topics of this dissertation. This work is finally concluded with an eye toward future research in Chapter 8.

CHAPTER 2

ORIGIN AND HISTORY OF THE PROBLEM

2.1 Software-Defined Networking

Understanding SDN conceptually typically begins by examining currently implemented components that enable programmability of a network. Networks that strictly adhere to SDN internally direct their packet traffic through very basic switches that examine the characteristics of incoming packets. These switches perform actions on the packets (forward, drop, modify, etc.) through their switch ports based on installed rules. These rules are defined, installed, and managed by a logically centralized process referred to as the controller. The controller must communicate with its switches in a standardized manner, and the predominant method for normalizing this communication at the time of this writing is the OpenFlow protocol[2].

2.1.1 The OpenFlow Protocol

The OpenFlow communication protocol is a popular standard under which SDN may be deployed. The protocol itself arose from the need for an effective method for analyzing and testing new protocols realistically and scalably. Such tasks were previously cumbersome and difficult due in part to the rigidity of currently installed networks and a hesitance to interfere with them at the risk of compromising the base network functionality. The OpenFlow protocol was introduced as an attempt to address these issues. It is an open protocol that enables researchers to run experimental protocols on large scale networks while maintaining the integrity of normal user traffic. With OpenFlow, the flow tables contained in modern Ethernet switches and routers are simplified to accommodate a general set of functions and can be programmed according to these functions.

An OpenFlow switch integrates a flow table, a secure channel, and the OpenFlow protocol. The flow table consists of a set of flow entries. A flow is a match qualifier linked with a list of actions to take if the specific match is found. Each flow entry in a flow table is composed of a set of packet fields to match and actions to perform, such as sending the packet out through a certain port, modifying some field or fields in the packet before forwarding it, or simply dropping the packet. Based on the requirements of a particular switch, it may reasonably contain more than one flow table.

The secure channel of an OpenFlow switch connects it remotely to the controller process. Across this connection, the switch and controller can communicate commands and data. This communication is standardized by the OpenFlow protocol which provides a means to interface with the switch without directly programming it. Establishing a connection between the controller and a switch requires a specific set of steps similar to most network protocol handshakes. Establishing the connection involves a number of message exchanges, beginning with an OFPT_HELLO message followed by other messages for acquiring and/or designating various switch features and configuration settings.

When an OpenFlow switch receives a packet for which it has no matching flow entries, it may send this packet to the controller through an OFPT_PACKET_IN message. Upon receiving this packet, the controller will determine the appropriate action for the switch to take. This action may either be performed a single time by the switch, or the controller may direct the switch to install a flow entry with the appropriate action with an OFPT_FLOW_MOD message. This entry will hold certain characteristics of the received packet to compare against subsequent packets. Referred to as an `ofp_match`, this set of packet fields can prompt the switch to perform a given action when similarly matching packets are received in the future. The addition of flow entries is accompanied by the ability for the controller to remove flow entries from a switch flow table. This removal may occur through a direct action sent to the switch by the controller or through timeout values held in the flow entry [2, 3].

Table 2.1: Packet match fields for OpenFlow 1.0

Ingress Port	Ether src	Ether dst	Ether type	VLAN id	VLAN prio- rity	IP src	IP dst	IP proto	IP ToS bits	TCP/ UDP src port	TCP/ UDP dst port
-----------------	--------------	--------------	---------------	------------	-----------------------	-----------	-----------	-------------	-------------------	----------------------------	----------------------------

Requirements formally defining the OpenFlow protocol may be found in the *OpenFlow Switch Specification*. At the time of this work, versions extend from 1.0.0 to 1.5.0 [4]. OpenFlow 1.0 was the first commercially viable version of the specification. The supported packet header fields, shown in Table 2.1, and actions are statically defined, necessarily limiting the scope of its functionality as SDN was still a relatively nascent concept at the time of its release. The only required actions in OpenFlow 1.0 are forwarding a packet through the physical and virtual ports of the switch or alternatively dropping it. Packets could optionally be enqueued, enabling simple QoS support. The packet fields from Table 2.1 could also optionally be modified.

The OpenFlow 1.3 specification[5], as the next version to gain widespread commercial use, expanded upon the functionality provided by OpenFlow 1.0 and other intermediaries to provide a more abstract and easily extensible protocol. As shown in Table 2.2, match fields are no longer static field types, as they are in OpenFlow 1.0, but instead described using the OpenFlow Extensible Match (OXM) format, comprising a type, length, and value (TLV) for each match field. The `oxm_class` designates whether a TLV is derived from the basic set standardized by the OpenFlow protocol (`OFPXMC_OPENFLOW_BASIC`). An `OFPXMC_EXPERIMENTER` class is provided as well, allowing researchers to test beyond the basic set of provided match fields. The `oxm_field` identifies the specific header field that will be used to match against the corresponding TLV. For example, in the `OFPXMC_OPENFLOW_BASIC` class, if the IPv4 source address is involved in the match, the appropriate `oxm_field` would be `OFPXMT_OFB_IPV4_SRC`. Using the OXM TLV format, the OpenFlow 1.3 specification is able to accommodate an additional

Table 2.2: OXM TLV header fields

Name		Size (bits)	Usage
oxm_type	oxm_class	16	Match class, reserved or member
	oxm_field	7	Match types within the match class
	oxm_mask	1	Set if OXM TLV contains a bitmask
	oxm_length	8	Length of OXM payload in bytes

28 possible match fields (depending on the type of flow traffic) compared to the 12 required by 1.0.

Another distinction between OpenFlow 1.0 and 1.3 is an additional layer of abstraction between flow matches and actions, referred to as flow instructions. These instructions provide a mechanism for either collecting or immediately executing actions such as the packet forwarding and field modification actions provided in 1.0. The addition of instructions enhances the level of control and specificity provided by OpenFlow. Additionally, a `GOTO_TABLE` instruction is included in the 1.3 specification. In the OpenFlow 1.0 specification, multiple tables are simply a tool to provide more capacity for more flow entries in their switches; when a flow lookup reaches the end of one table, it simply moves to the next one. However, in OpenFlow 1.3, reaching the end of a table implies that the lookup has completed the process of collecting actions and is ready to execute them. Moving to another table only occurs due to a `GOTO_TABLE` instruction. In this way, tables can be assigned specific purposes instead of simply acting as extra space.

2.1.2 Controller Libraries

A logically centralized controller communicates with SDN-enabled switches to which it is connected. This controller is a process which sends rules to the switches to determine the forwarding behavior that they implement. Currently, these rules are typically structured based on the different versions of the OpenFlow protocols, depending on switch compatibility. Newer ideas have revolved around the premise of making future switches protocol-agnostic. Numerous controller frameworks have been developed in various languages, such

as C++, Python, Java, and even Erlang. Choosing a particular controller library for SDN deployment ultimately depends on the needs and specifications of a particular organization. Some options studied in this work include the Python-based libraries Ryu[6] and POX[7] and the Java-based Floodlight controller[8].

POX

POX is the Python-based variant of NOX, the first controller library to support OpenFlow. POX provides a means for writing “Pythonic” controller applications compatible with OpenFlow. As a framework written in Python, its installation requirements are relatively low. Additionally, it possesses a low learning curve due to its design, and as such, has seen significant adoption in the academic realm as a mechanism to introduce SDN to the broader community. POX provides its own coordination library dubbed *recoco* that handles threading, timing, and other synchronization tasks. Accompanying the POX framework are numerous built-in controller applications that perform some basic forwarding functionality such as MAC address learning, link layer discovery, spanning tree, ARP requests, etc. At the time of this writing, POX supports OpenFlow 1.0[7].

Ryu

Ryu is a modular SDN controller framework written in Python. In terms of compatibility, it is a more robust framework than POX, supporting fully OpenFlow versions 1.0, 1.2, 1.3, 1.4 and 1.5. Additionally, it supports the configuration protocols OF-config and Netconf. It provides OpenStack support and is considered well tested by its creators. It is also highly regarded by the SDN community as a commercially ready option within an SDN deployment. Within its framework, Ryu handles its synchronization and coordination efforts using the *eventlet* Python library. Event callbacks in Ryu applications are handled by decorators that specify which events trigger them, such as packet in or port status messages, and at which times the function may be called. Ryu provides many similar built-in applications

as POX while also including a RESTful API as part of its OpenStack support[6].

Floodlight

Floodlight[8] is a modular, Java-based SDN controller that can be adapted and configured either manually or through a RESTful API. It supports both virtual OpenFlow switch libraries, OpenvSwitch and *ofsoftswitch*, as well as nearly 30 hardware switch models at the time of this writing. Packaged and handled as a single JAR file, it can be immediately booted to implement its default Forwarding application, which provides basic connectivity between the devices connected to the switches it is tasked to control. By default, the controller includes topology maintenance, load balancing, and basic end-to-end routing. Floodlight fully supports OpenFlow versions 1.0 and 1.3 while also allowing experimental compatibility of versions 1.1, 1.2, and 1.4.

2.1.3 OpenvSwitch

OpenvSwitch[9] is an open-source, OpenFlow-enabled virtual switch library primarily aimed toward delivering efficient networking capabilities in the rapidly expanding and resource-demanding environment of data centers and similar infrastructure. It is composed of the `ovs-vswitchd` daemon which resides in userspace and a *datapath kernel module* that operates at the lower-level kernel space. Under this design, incoming packets to the switch are first received by the datapath module. If the received packet has a matching action already installed on the switch, it can make any necessary header field modifications and then simply forward it out the appropriate switch port without interacting with the userspace. However, if an action is not found in the flow tables of the switch, it is forwarded up to `ovs-vswitchd` in the userspace, where it either internally determines how to handle the packet or forwards it to an external controller and awaits instructions. Within the datapath in kernel space, OpenvSwitch caches its OpenFlow tables and makes the distinction between fine-grained *microflows* with many packet match fields and less specific *megaflows*

when determining how to adequately handle certain traffic flows. Notably for this work, the addition or removal of certain packet headers such as VLAN tags and MPLS labels cannot be accommodated efficiently in kernel space and must instead be solely handled by `ovs-vswitchd`. This caveat begets a potential performance drawback compared to flow traffic that does not require these packet modifications. However, it also facilitates immediate and practical modification of the OpenvSwitch library, a convenience that is not as readily permissible in kernel space.

2.2 Direct Code Execution

Communication networks are constantly evolving with new technologies aimed at providing greater quality, resiliency, and security. Modeling and simulation provide an avenue for examining the traffic within new or existing network topologies. In simulating a network, characteristics of the topology may be derived without interfering with the existing framework or incurring an immediate hardware or software cost. Popular network simulators such as ns-3 are effective tools for studying these network behaviors. However, adequate coverage of new network programs and protocols within simulation requires porting models of these new applications into the simulators. These efforts require a significant amount of time for development and validity testing against real-world behaviors. These drawbacks can be especially pronounced for SDN due to the lack of portability in modeled controller applications. Instead, a mechanism for directly deploying real-world network applications within a simulated environment can alleviate these issues while adding realism. The Direct Code Execution (DCE) framework in ns-3[10] provides the capability to execute user space and kernel space network protocols and applications directly within an ns-3 simulation. These applications typically require no source code modifications. DCE interacts with the installed binary similarly to how an actual operating system would.

DCE is composed of three layers: the core, the kernel, and a POSIX layer. The *core* layer provides virtualization mechanisms to coordinate the actions of the simulated pro-

cesses within the context of the ns-3 event scheduler. At this layer, global variables, threads, stack space, and the heap for each particular process are all managed. The *kernel* layer connects the real-world Linux TCP/IP stack to the simulated physical layer (L2) of ns-3. Traffic from the installed applications can then traverse this hybrid simulated stack from application-level socket function calls. The POSIX layer of DCE replaces the real GNU C Library (*glibc*), the standard library for the C programming language. Calls to *glibc* functions are caught by this layer to determine how they should be handled. In many cases, DCE does not need to manage certain *glibc* calls, such as `math` or `string` functions, and can simply pass the call to the *glibc* of the underlying system. Time-related functions return information about the simulated time rather than the wallclock time. Socket function calls are effectively wrappers to the simulated sockets for the ns-3 stack. Subprocess and threading functions are also handled by this layer to manage their contexts. Local files are managed in the POSIX layer relative to specific file space for each node running DCE applications.

DCE has previously been used in numerous experiments to enhance the realism of network simulations. Demonstrations of DCE in literature have primarily focused on the fields of mobile and wireless networks where updated protocols are introduced frequently as the technology improves. Rather than constantly porting and modifying these procedures into simulation, it can be much quicker to simply introduce these protocol implementations into simulation via DCE. In [11], a comparison of the ns-3 Optimized Link State Routing (OLSR) model with an actual OLSR daemon in DCE uncovered deficiencies in both programs. Addressing tuning issues in the simulated model and the daemon program allowed them to be updated to better fit the OLSR RFC. Demonstrations of content-centric networking (CCN) over mobile ad-hoc networks (MANETs) and multipath TCP over LTE and wireless in [12, 13] provide examples of additional use cases for DCE that required no modifications to the original implementations.

2.3 SDN Simulation/Emulation Tools

When modeling systems such as communication networks, simulation and emulation aim to provide adequate representations of the behaviors of these systems. However, simulation time is not intended to coincide with wallclock execution time. It is preferred that simulations cover a greater simulated time span than the runtime execution time span. Emulation is intended to execute as closely as possible with the wallclock time, typically to interact with the real world in some way. Furthermore, work in [14] demonstrates how emulation can be ill-suited for large-scale SDN evaluation. Accurate control plane emulation, unpredictable update time ordering, and limitations related to scaling down a representative topology through bandwidth shrinking are just some of the drawbacks noted for SDN emulation.

Mininet[15] is a network emulator that employs virtual Ethernet pairs and processes in network namespace in Linux to allow lightweight, rapid prototyping of SDNs. Through its Python-based API, scripts can be written to construct custom topologies of switches and hosts. Network traffic from the hosts may then be controlled and monitored through API commands to analyze the correctness and performance of the network. The hosts in Mininet are simple shell processes that are given their own network namespace. These hosts are given a virtual network interface and are children to the main Mininet process. Software switches supporting OpenFlow are given their own virtual interfaces as well and forward packets through the virtualized topology based on instructions from SDN controllers. The SDN controllers may exist internally or externally to the process space in which Mininet is run. As long as the system on which Mininet is run has network layer connectivity, the network interfaces of the virtualized switches can be configured to communicate with the controller process, regardless of its location. The links between all of these components (hosts, switches, controllers) are virtual Ethernet pairs, or *veth* pairs, that act as wired connections.

Mininet provides a network testbed similarly to simply creating collections of virtual machines (VM) and connecting them in a specific topology. However, without the overhead of entire computer systems (OS, memory, etc.), Mininet is able to achieve similar results with significantly fewer resources. Even with the lightweight features that Mininet utilizes, it still presents some limitations related to performance and resource usage. All components in a topology created by Mininet require their own process space. At small scale (hundreds of nodes or fewer), performance impacts are not necessarily incurred since the memory usage of the system is not fully exploited. However, at scales of thousands of nodes and more, performance can significantly degrade as more resources are required to fully realize the underlying components of each node in the topology. A workaround is available in the form of CPU limited hosts that only use a specified percentage of the total process space. With greater numbers of nodes though, this design choice can impede the realism of the virtualized hosts. Additionally, the use of CPU limited hosts only addresses memory usage for the hosts. Limitations are not imposed on the virtualized switches and internal or remote controllers so these components will use as much process space as they would typically need. The emulated default connections in Mininet also present a performance fidelity issue as the *veth* pairs will not provide specific bandwidth limits or quality of service. For cases where additional characteristics need to be added to the links, TCLinks that employ the Linux traffic control (*tc*) program may be used. Even so, the main Mininet process is still obligated to operate under the Linux scheduler of the system on which it is run. In this way, the emulated nature of Mininet will not guarantee identical results as simulation would. Limitations regarding scale, quality of service (QoS), and performance fidelity have been examined[16, 17].

The flow simulator *fs*[18] is a flow-level discrete event network simulator written in Python. Instead of operating on packets, it operates on the higher-level notion of a *flowlet* as its network abstraction, grouping streams of packets between a sender and its recipient. The number of events processed within an *fs* simulation can be reduced compared to

packet-based simulation as fewer events are required to model groups of packets than each individual packet. With fewer total events to process, the simulation in *fs* is typically able to complete in less time but with less precision than a comparable packet-based simulation. SDN capabilities have been introduced into the *fs* platform in an extended framework referred to as *fs-sdn*[19]. This framework is capable of directly incorporating the POX OpenFlow controller libraries and API without modification. Because the underlying *fs* system works at the flow level rather than the individual packet level, interfaces between *fs* and POX deconstruct flowlets into individual packets and vice versa. *fs* was designed to operate at the network layer (IP), so it is incomplete for handling lower layer protocols. Further work on *fs-sdn* has focused on making the framework controller-agnostic[20].

The network simulator ns-3 provides simulation/emulation frameworks for developing network topologies and analyzing their network characteristics. It is developed in C++, and its libraries may be accessed in the same way or through Python bindings. Both formal and informal attempts have been made to provide varying levels of SDN capability in ns-3. An `OpenFlowSwitchNetDevice` class was added to the ns-3 baseline in ns-3.11[21]. The implementation of this class suffers from two main drawbacks. The OpenFlow specification that it supports is 0.8.9, an early, experimental version that is not intended for commercial use. Additionally, the interface of the `OpenFlowSwitchNetDevice` is designed with a built-in controller as an embedded component of the implementation. This design prevents examination of controller applications written for real-world libraries. It also hinders proper link testing between controllers and switches in terms of topology checking and traffic verification. Alternatively, ns-3 provides mechanisms for allowing the simulated topology in the ns-3 process to interact with real-world network components. One of these mechanisms is the `TAPBridge`. Using the `TAPBridge`, packets sent from a simulated node in ns-3 may be sent out of the simulation to a real-world recipient. Incoming packets may be tunneled through the `TAPBridge` and received within the simulation. With the `TAPBridge` capability in hand, links between the components in a Mininet topology may

be constructed based on channels provided by ns-3[22]. OpenNet[23] extends this effort to simulate software-defined wireless local area networks (SDWLAN).

Efforts in the discrete event simulator OMNeT++ provided OpenFlow components based on OpenFlow version 1.0.0[24]. These components were built using the INET framework. Successful implementation was demonstrated through an evaluation of mean round trip time (RTT) for the simulation of a particular topology. However, the project designed its own controller rather than allowing for the use of external controller libraries such as POX. This design lacks controller application portability, making it difficult to compare its performance against hardware testbeds.

The Estinet simulator/emulator[25], a commercial product based on the network simulator NCTUns, provides one of the more well-rounded options for SDN simulation in that network traffic and topologies may be configured either textually or through a GUI. The Estinet design employs a mechanism for reentering the kernel. This process allows it to support the simulation of NOX, POX, and Floodlight controller applications with complete portability. However, as proprietary software, it is difficult to accommodate for a complete performance analysis as the source code is not distributed with the product. This issue prevents a proper examination of the complete implementation of the Estinet design. The correctness, scalability, and performance of Estinet is compared against Mininet in [26], but this work only examines a single grid topology under simple use cases.

This work specifically analyzes simulation/emulation frameworks that can be executed on a single machine. However, other frameworks have been designed to allow simulations to be executed in a distributed environment to accommodate additional topology and traffic scale. Mininet CE[27] allows Mininet networks on different machines to be run collectively as a single emulation. The Distributed OpenFlow Testbed (DOT)[28] emulates a network across multiple computer systems. Differing from the clustered design of Mininet CE, DOT designates one computer system as a DOT Manager that controls the other involved systems, referred to as DOT Nodes. Otherwise, DOT implements its emulations similarly

to Mininet with *veth* pairs.

2.4 GENI

Similarly to the emulators described in Section 2.3, virtual network testbeds, such as the Global Environment for Networking Innovation (GENI), provide the opportunity to perform network research on actual hardware. In contrast to the emulators which run on resource-constrained single systems or at limited distributed scales, these testbeds provide ample compute resources which in turn produce more realistic results. GENI is a national federated hardware testbed that provides these virtualized networking resources. Universities and other research institutions have collaborated to provide an environment that can be provisioned and leveraged for use in research in SDN and other cutting-edge networking paradigms. Participating campuses contribute functional components to GENI through the use of GENI racks and SDN. The resources located at a particular campus are referred to as a GENI aggregate. The GENI racks act in much the same way as traditional servers in data centers and the “cloud” by allocating resources such as VMs in a fair and reliable manner. Using SDN, numerous distinct networks can be realized across GENI racks through virtual network slicing. The resulting slices are isolated from one another, guaranteeing the integrity of each created network in terms of its functional characteristics such as its link parameters while preventing interference across slices. Furthermore, a diverse array of resource options are available through specific GENI instantiations: PlanetLab[29] resources can be obtained via linux Vserver virtualization; kernel-based VMs (KVMs) are available via OpenStack in ExoGENI racks; InstaGENI and ProtoGENI provide OpenVZ and XenVM resources; and even raw physical hosts can be requested from ProtoGENI. Link resources can be varied through a number of tunneling techniques. Within a single GENI aggregate, compute resources may be connected and configured as a single link or a broader LAN. Across multiple aggregates, connections must be created as either Stitched Ethernet links or Layer 2 or 3 Generic Routing Encapsulation (GRE) tunnels. These links

provide a shared 1Gbps bandwidth by default, while lower rates can be set and higher ones are discouraged.

Deployment of a GENI experiment involves the following steps: resource specification, virtual allocation, experimentation, and resource release. Specifying the resources that are required for a particular experiment is handled through an XML-based resource specification (*rspec*) file. This file describes the type and number of resources to be requested and how they are linked together while also permitting more fine-tuned installation behavior. Software installation, service initialization, and other scripted behaviors can be defined for particular compute resources while bandwidth, delay, and other link characteristics can be configured for the network connections. Describing the resources to be requested through a *request rspec* can be accomplished via multiple interfaces. The Jacks tool provides a graphical interface for designating resources. The *omni* command line interface (CLI) supplies another method for interacting with an *rspec*. Additionally, the *geni-lib*[30] Python API allows users to create a request *rspec* through a Python script. With a request *rspec* created, a user can submit it to one GENI campus (or multiple if it is configured as such), and if sufficient resources are available, the GENI campus (or campuses) will respond with a *manifest rspec* detailing the specific resources that have been allocated. As these resources are real as opposed to simulated, they require some time to boot and perform any requested installations or services. Once available though, the GENI experimenter can perform any required experiments, analyzing various SDN controller or switch configurations and generating traffic as simple as *ping* or *iperf* or through various freely available packet generators. Resources are requested for a certain amount of time but may be renewed as demand permits. Upon completion of experimentation, the resources can be returned for use by other experimenters.

2.5 Routing in Simulation

In network simulations, packet forwarding should result in routing behaviors resembling those that would occur in real-world topologies. The most direct way to accomplish this goal is to implement models of routing protocols, such as BGP or OSPF, that would maintain the routing tables for each simulated node in a topology. However, this design is not scalable as it incurs significant memory overhead managing routing tables for every node in a simulated topology. Instead, it can be more efficient to manage the routing decisions for these simulated topologies either collectively, reducing some of the redundancy that multiple routing tables would observe, or on-demand, determining routes as necessary. Collectively computing routes can be accomplished through any shortest path graph algorithm. However, it can require substantial initialization time as all routes are globally calculated, and in the worst case, still incur significant memory usage. These issues can be minimized though if route calculation is accomplished in a parallel manner using the massively parallel processing capability of graphics processing units (GPUs). Using a parallelized version of the Floyd-Warshall algorithm, the work in [31] has successfully demonstrated this more efficient mechanism for route generation.

On-demand route determination has been examined using *Neighbor Index* (NIX) vectors[32]. These NIX vectors collect next-hop information at each node along a route until the destination is determined. In simulation, this mechanism can provide stateless routing, reducing routing table storage and processing time[33]. It can also be made more resilient to topology changes by maintaining a record of when each NIX vector was last known to be valid[34]. A variant, referred to as *MTree Nix*[35], has demonstrated an 85% reduction in simulation time compared to traditional NIX vectors by combining NIX vector routing with a minimal routing table based on multiple spanning trees. NIX vectors have also been demonstrated in mobile ad hoc networks[36], where they exhibited lower latency and better throughput than Dynamic Source Routing (DSR).

2.6 General-Purpose Computation on GPUs

General-purpose computation on GPUs (GPGPU) is a programming paradigm that exploits the massive parallel processing capability of GPUs to handle the simultaneous computation of large amounts of data. The Compute Unified Device Architecture (CUDA) is a general purpose parallel computing platform and programming model that enables GPGPU for NVIDIA GPUs. Programming with CUDA allows the GPU to act as a coprocessor with the ability to spawn a large number of parallel threads. Code that will execute on the GPU is known as a kernel. It must be compiled to a GPU specific binary. When a program executes, its compiled kernel is pushed to the GPU along with the data that it needs to process. Kernel invocation from the CPU, in addition to specifying the function that will be launched, includes parameters for a programmer-provided *grid*. A grid is composed of a *block count* – the number of blocks it intends to use – and a *block size* referring to the number of threads in each block. The most threads that a block may contain is 1024 in current hardware. It should always be evenly divisible by 32 based on how the threads are executed on the hardware.

GPUs are designed around arrays of multi-threaded *streaming multiprocessors* (SM). Each SM in the NVIDIA *Kepler* architecture is composed of 192 single-precision cores and 64 double-precision cores. Each SM also possesses 32 special function units and 32 load/store units. The special function units can support operations such as sine, cosine and square root. Upon execution, each grid block is given to an SM until it completes. Each SM can hold up to 16 blocks or 2048 threads, based on whichever limit is reached first. Thread execution occurs in *warps*, groups of 32 threads within an SM. This concept implies that block sizes should be evenly divisible by 32 in order to completely fill allocated warps. Otherwise, some warps will hold threads that do not do anything. Threads are given sequential thread identifiers within their warp. Warp threads are executed similarly to the *single instruction multiple data* (SIMD) pattern in Flynn's taxonomy[37]. However, standard SIMD

suggests that every thread will execute every instruction whereas the CUDA model allows programmers to insert divergent code paths into their kernels. This mode of execution is considered *single instruction multiple thread* (SIMT). At the hardware level, certain threads will be either enabled or disabled based on their response at the point of divergence. From this point until the paths reconverge, some threads will execute instructions while others will effectively `noop`. When handled correctly, divergence provides additional flexibility when designing a kernel. However, significant performance degradation can occur when divergence is not designed well.

Interacting with an NVIDIA GPU beyond simply spawning kernel functions occurs through one of two APIs, the CUDA driver or runtime libraries. Both packages provide functions for moving data between host and device, querying characteristics of available GPUs, managing their threads and streams, etc. The driver API provides a lower level interface allowing for more fine-tuned control of the interactions with the GPU. However, it requires the programmer to maintain contextual information about the GPUs in use. Additional configuration is also needed to prepare kernel functions for launch. In contrast, the CUDA runtime API hides many of the lower level details, internally managing context and hiding kernel configuration.

2.6.1 GPUs for Networking and SDN

GPUs have been examined in networking environments to leverage their massive multi-processing capabilities for highly parallelizable tasks. Traffic monitoring and analysis has benefited from GPU acceleration in [38]. In [39], a scalable GPU-based IP lookup engine is introduced that maintains stable throughput and latency. An architecture for accelerating packet classification through parallel means is described in [40], building on a parallel classification algorithm called gPF[41] that is optimized for CUDA devices. Prior work has evaluated the efficacy of parallel versions of pattern matching algorithms for network security. In [42], a parallel version of the Aho-Corasick algorithm is employed using a

GPU. String and pattern matching algorithms for intrusion detection systems (IDS) are designed and studied in [43] and [44] as well. Encryption is another field in network security that can benefit from GPU acceleration with [45] demonstrating a parallel algorithm for the Advanced Encryption Standard (AES). Each of the studies mentioned relied on either parsing previously recorded data or deploying actual hardware to evaluate them. These efforts would have benefited greatly from a suitable simulation framework.

More specifically, research efforts have also focused on adapting certain highly parallelizable tasks in SDN environments to operate on GPUs. Most studies have focused on accelerating flow entry lookups and packet/string matching and classification on software-based switches, such as those running the OpenvSwitch kernel[46, 47, 48, 49]. Other studies have examined applicable uses of GPUs in SDN for security, either accelerating packet classification for intrusion detection controller applications[50] or packet encryption on middleboxes[51]. A GPU SDN controller[52] has been designed that handles incoming switch events in parallel on a GPU when the workload achieves a certain threshold.

CHAPTER 3

ENABLING NETWORK SIMULATION TO SUPPORT REALISTIC AND PORTABLE SOFTWARE-DEFINED NETWORKING CAPABILITIES

3.1 Extending DCE Beyond C/C++

The concept of direct code execution as discussed in Section 2.2 is primarily constrained to network applications and protocols written in the programming language of the employed simulator. For DCE in ns-3 and some similar network simulators, prior work has demonstrated their effective use for applications written in C or C++, which is conveniently accomplished because these simulators are written in C++. The Python-based flow simulator *fs* includes extensions that can connect it to external applications simply because those applications are also written in Python[19]. The efforts in [53] introduce a framework for allowing DCE to accommodate and execute code for network applications written in languages other than C or C++, specifically Python and Java. The executables that launch applications written in these languages are simply native code binaries that are built from C/C++ source code. This fact is exploited in order to launch Python and Java applications within the DCE environment, allowing their source code to be interpreted line by line from within the context of the simulation. In this way, entirely new sets of network applications can be examined through the DCE environment without source code modification or the need to translate the program to C/C++.

3.1.1 Programming Languages

This section provides a simple technical overview of the programming languages examined in this work. It begins by examining the languages for which DCE was originally intended to be compatible, namely C and C++. Through extensions for DCE performed in this work,

two additional languages have been made operable within the ns-3/DCE environment. The interpreted language Python and its predominant runtime library CPython are described. Additionally, the Java programming language and its runtime environment, the Java Virtual Machine (JVM), are discussed.

C/C++

The C programming language is considered one of the first mainstream, general-purpose programming languages. It is a statically typed, procedural language that has been adopted for use in a variety of systems. It is one of the “lower” high level languages available with many newer languages, including C++, Java, and Python, employing it as an intermediary at some point in their respective compiler/runtime pipelines. As stated in section 2.2, the standard library for the C programming language, referred to as *glibc*, provides a substantial level of functionality. String and memory manipulation, mathematical functions, system time information, file and socket handling, parallel processing, and a variety of other capabilities are available in *glibc*. The C++ programming language was originally intended as an object-oriented enhancement to C. In addition to similar features as C, C++ enables class creation complete with abstraction, encapsulation, inheritance, polymorphism, templates, and operator overloading. A standard API is provided by the C++ Standard Library, which *glibc* currently supports. Both C and C++ are compiled to native code that an underlying computer system can recognize equivalently. This characteristic results in them being effectively identical from the viewpoint of the DCE environment.

To operate as DCE applications, programs written in C or C++ must be recompiled such that DCE recognizes them as dynamic libraries rather than static executables. In this way, DCE can load the `main` function of a particular program as if it was just another addressed symbol in the memory space of the dynamically loaded library. Compiling the source code into position-independent code with the `fPIC` flag allows it to be loaded similarly to a shared object library. Subsequently, linking with `pie` and `rdynamic` re-

spectively produces a position-independent executable and ensures that all symbols of the resulting executable will be loaded into the dynamic symbol table. Regarding operability within DCE, the addition of these flags to the compile and link steps is generally the only modification required for building a target application for execution in DCE. However, additional considerations may be necessary if certain functions used by the application are not currently implemented in the *glibc* coverage of the DCE baseline.

Python

The Python programming language is a multi-paradigm, general-purpose, high-level language that aims to be more readable than other general-purpose, object-oriented languages, such as C++ or Java. Today, programming in Python is available in two versions: Python 2 (most recently 2.7) and the backwards-incompatible Python 3 (currently 3.5). The reference implementation for both versions is CPython, whose source is written in C. CPython acts as a source code interpreter more so than a compiler. When Python programs (or *scripts*) are provided as inputs to the `python` command, the code will be read and directly executed by the Python runtime. This fact suggests that, with the proper configuration, the `python` command can be built to operate in the DCE environment. In this way, when it is provided a Python script, the source lines will be interpreted, and the underlying *glibc* calls can be handled by DCE. The focus of this work is the CPython implementation of Python 2.7.

The CPython source code did not require any modifications. In building it into the `python` executable and libraries, it required the typical flags such that DCE could recognize and load it, i.e. `fPIC` for compile and `pie` and `rdynamic` for linking. Similarly to *glibc*, Python is equipped with the Python Standard Library (PSL), a set of objects and APIs written in Python that provide a standardized baseline of Python programming capabilities. Because the PSL is written as Python scripts and not a “built” library in the same sense as *glibc*, it is loaded in a different way. A Python script is generally “imported” into

the Python runtime through the `python` command. In the context of DCE, this step occurs after the `python` executable is loaded and executed. In this way, the effective kernel space has transitioned into simulation. At this point, the location of the user-defined Python application and baseline Python scripts in the PSL such as `os`, `socket`, and `string` must be placed where the DCE-enabled node can “see” them. To make the files visible to the simulated node, the user-defined script is copied into the filesystem for the node, and the PSL is symbolically linked under this filesystem as well.

Java

Java is a general-purpose, object-oriented programming language similar in some ways to C++. Java applications are compiled but not to native code. Instead, they are converted to Java bytecode which can be executed on the JVM. The JVM along with the standard Java Class Library (JCL) comprises the Java Runtime Environment (JRE) which provides the APIs and executable environment for running Java programs. When running a compiled Java program, the JRE will initialize the environment, and then the JVM will interpret the provided bytecode into native code that the underlying system can understand. One version of the JVM, HotSpot, provides performance optimizations such as adaptive compilation as well as efficient heap management and garbage collection. Development of Java programs is enabled through the Java Development Kit (JDK), which allows the applications to be compiled and packaged.

The OpenJDK library, an open-source implementation of the Java Standard Edition (SE) Platform, provides a configurable mechanism to build an interface between DCE and the Java programming language. The JRE and JDK provided by Oracle are only distributed as binaries such that they are only available in specific build configurations. In contrast, OpenJDK is available as source code that can be built – compiled and linked – with the position-independent and dynamic flags that allow DCE to load its programs into simulation. The “program” of specific interest is the `java` command. The source code that

produces the command and the JRE and JVM libraries that it calls to configure and execute the Java environment are all written in C. In this way, applications written in Java that are compiled to class files that the `java` command will accept will ultimately be interpreted to *glibc* symbols. These symbols can then be loaded and executed by the DCE environment. The implementation of OpenJDK used by this work is OpenJDK 8.

The JRE, designed around the JVM, requires additional considerations beyond simply addressing *glibc* symbols that the DCE baseline has yet to include. This process did require the inclusion of additional symbols, some related to determining the process location for newly created Java threads within virtual memory. However, accommodating the JVM also needed to address determining networking interfaces for the DCE-enabled nodes. Under the baseline OpenJDK source code, information about network interfaces and next-hop routes for a system running the JVM is gathered from the `/proc/net/if_inet6` and `/proc/net/ipv6_route` files, respectively. (The JVM handles translating addresses between IPv4 and IPv6 schemes.) The information in these files is relatively easy to gather. However, an interesting issue presents itself in the form of buffer limitations for standard I/O methods such as `sscanf` within the DCE environment. Limitations are implicitly imposed by DCE on the applications it manages simply due to the nature of handling virtual kernel space inside of a simulation. To model the environment the JVM expects as closely as possible, filesystem for a DCE-enabled node is created with the `/proc/net` directory in place. However, to accommodate the noted buffer issues, modified versions of the `if_inet6` and `ipv6_route` files are written that only hold the information that the JVM needs. For `if_inet6`, this information is the hexadecimal representations of the IPv6 addresses for a node as well as their indexes and device names. The `ipv6_route` file that is responsible for configuring routes for the node holds the base IPv6 addresses as 32-character hexadecimal values, the hexadecimal prefix length (similar to the IPv4 subnet mask), configuration flags, and the device names.

3.1.2 Experiments

Three sets of experiments have been performed to confirm – and in the case of C/C++ applications reaffirm – the relative range of functionality DCE provides for a variety of applications. Since both C and C++ programs would compile to roughly similar native code representations, only C programs have been examined. Alternatively, the choice of compiler between the GNU C Compiler (GCC) and the Clang frontend for LLVM is introduced for examination to gauge potential differences in compiler optimizations. In one set of experiments, a single node is tasked to perform some simple applications written in each language. In the second set of experiments, a simple dumbbell topology tests basic networking functionality for client/server-style applications written in each language. The final experiments task a host to ping every other end host in a ring topology multiple times to examine the scalability of the simulations when handling multiple DCE applications. All experiments have been performed using ns-3.24.1 and a modified version of DCE 1.7.

Performance Benchmarks

The first set of experiments installs programs written in C, Java, and Python on a single DCE-enabled node. These programs provide computational workload through some simple algorithms to confirm basic language capability in terms of data types and operators. They also test simple functionality in threading and local (loopback) networking, two relatively common features in networking applications.

- Matrix Multiplication (MatrixMult): 1000-by-1000 array matrix multiplication.
- Calculate π (PiDigits): Display 100,000 digits of π .
- Simple threading (ThreadTest): Create 500 threads that print 1,000 strings.
- Local Ping (Ping1000): Perform an Internet Control Message Protocol (ICMP) echo request 1,000 times to the *localhost* address and receive ICMP replies.

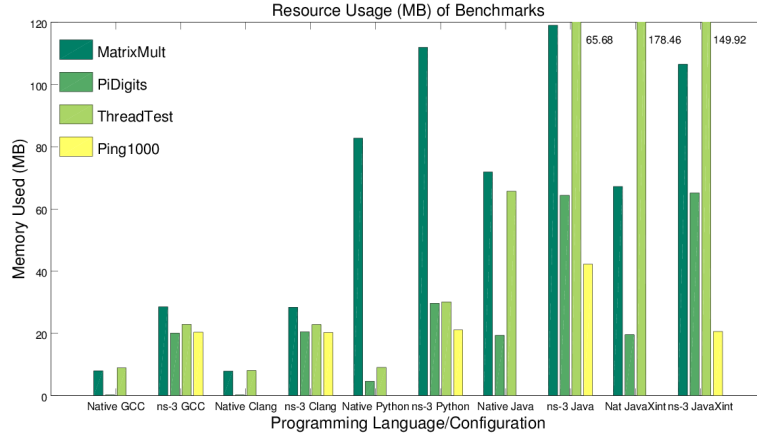


Figure 3.1: Single-node memory usage

Performance benchmark total memory usage in MB for a single DCE-enabled node and native equivalents for the various programming languages, compilers, and configurations. All data have standard errors that are less than 2% of their reported values.

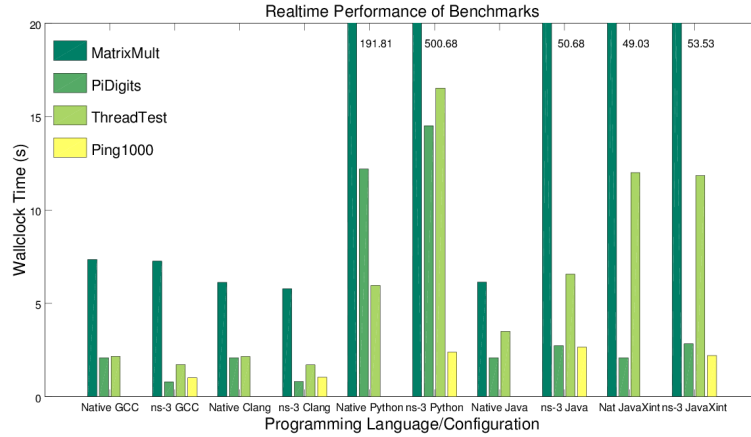


Figure 3.2: Single-node wallclock execution time

Performance benchmark wallclock execution time for a single DCE-enabled node and native equivalents for the various programming languages, compilers, and configurations. All data have standard errors less than 2% of their reported values except the GCC and Clang versions of the simple thread test. Their standard errors are 6% of their values.

The benchmark results are shown in Figures 3.1 and 3.2 alongside results of each application executed natively in the real Linux environment rather than the ns-3/DCE simulated environment. (The local ping benchmarks are not compared against native versions due to timing differences in the application designs in each programming language.) Java is run both regularly and with the `-Xint` flag to allow it to run in interpreted mode without some of the “performance benefits” of the JVM. The programs compiled with GCC and Clang required roughly the same amount of memory and produced similar time results. Based on the programs used in the benchmark simulations, the resource results are as expected. Most of the variables in the C programs had been allocated with stack memory and relatively few were needed, providing little room for any significant compiler optimizations. Based on the notion that both Python and Java are effectively invoking C/C++ calls within their underlying libraries, a certain amount of computational overhead is expected. In most of the benchmark tests, a tradeoff appears between lower memory usage with higher wallclock times in Python versus higher resource requirements with quicker execution performance in Java. The printing of the digits of π as well as the threading test produced a relatively significant discrepancy between resource usage and execution timing. However, this tradeoff is actually not realized for the matrix multiplication program. This result suggests that dynamically typed `list` allocation and assignment for the Python arrays incur both a resource and performance cost compared to the statically typed integer array in Java. Additionally, within the DCE environment, the “performance benefits” of the JVM appear to provide some level of speed-up to interpreted-only Java. However, these benefits come with increased memory usage in some cases. Threading appears to be one of the few cases that benefits in terms of memory and time from full access to the JVM. The local ping results are relatively flat compared to the other benchmarks most likely implying that the overall program ultimately lacked a significant amount of processing.

Simple Topology

A simple dumbbell topology is constructed to confirm successful networking capabilities are achieved in C/C++, Java, and Python as shown in Figure 3.3. Network interfaces for the links are configured on different 255.255.255.0 subnets (CIDR /24) to confirm packet transmission occurs through successful L3 routing and not simply ARP requests. Both end hosts in the described topology are enabled for installation of DCE applications. One end host acts as a client pushing data to the other end host. The client establishes a connection through a TCP socket with the other end host. The client allocates 65,536 bytes for transmission. This amount is selected to ensure that the socket connection accommodates multiple segments (based on the ns-3 default segment size of 536 bytes) without encountering congestion control, a process of the simulated stack rather than the examined program. These allocated bytes are transmitted in iterative chunks to examine the processing overhead of socket writes for the different programming languages. Upon completion of the specified number of transmission iterations, the client program exits. At the other end host, a hybrid server/client program is installed. The server aspect of the program listens to a TCP socket on the same port for the client on the other host to connect. When the client makes a connection to the server, the server program accepts the connection and reads bytes from the socket until the client closes the connection. Upon closing the initial socket connection, the server program stores the number of received bytes and relays the same amount of bytes back to the first end host on a new TCP connection. Ready to accept packets on this new port is an ns-3 PacketSink application. The server-turned-client iteratively transmits 65,536 bytes in 1,024-byte packets.

Simple dumbbell topology results are graphed in Figures 3.4 and 3.5. For the C applications, memory remains approximately constant as address space is simply and explicitly reused while performance intuitively requires more processing time as more processing is required. The Java versions of the programs had to be run in interpreted mode in order to complete. Full usage of the JVM is prevented due to its alternation between periods

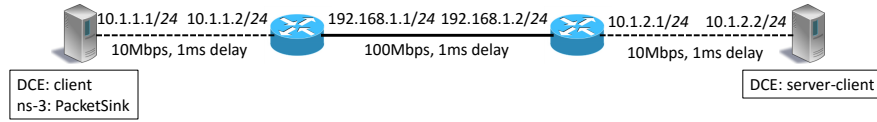


Figure 3.3: Simple dumbbell topology

The simple dumbbell topology is used to confirm successful socket handling and data transmission and reception for C/C++, Java, and Python. The topology consists of 4 nodes connected linearly with the outer 2 nodes acting as end hosts and the inner 2 acting as routers.

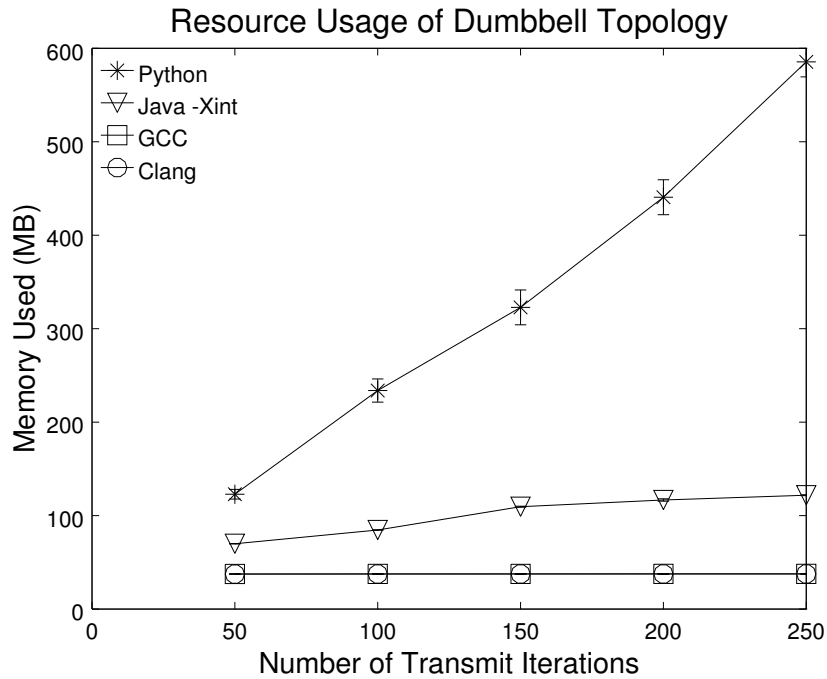


Figure 3.4: Simple dumbbell topology memory usage

Simple dumbbell topology results for total memory usage in MB. Standard error bars are displayed for each data point.

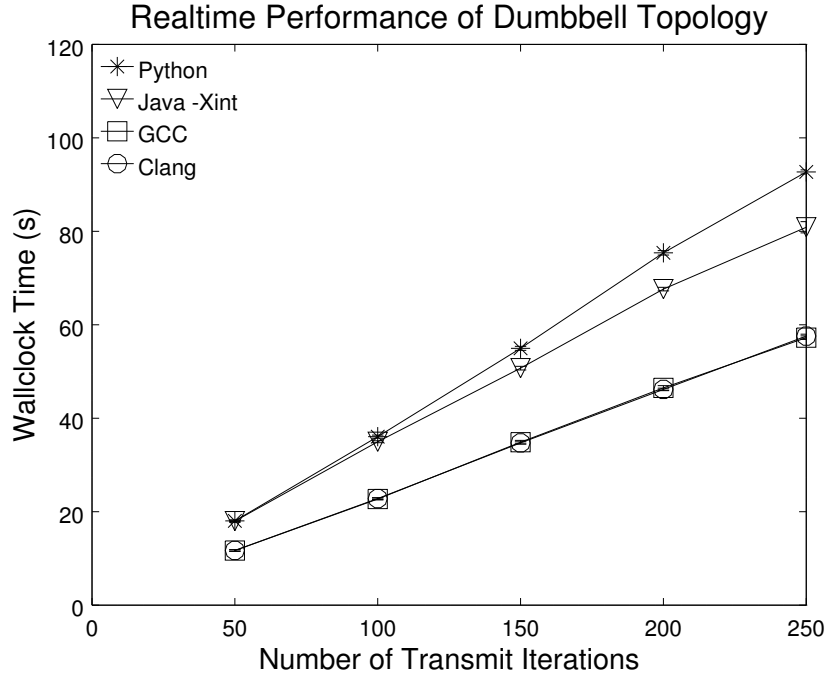


Figure 3.5: Simple dumbbell topology wallclock execution time
Simple dumbbell topology results for wallclock execution time. Standard error bars are displayed for each data point.

of optimization and deoptimization for profiling and debugging. In testing the framework, it was determined that this feature actually presented conflicts for the addressable space that DCE maintains. Again, an overhead is noted for the interpreted Java and Python programs compared to the C versions. However, interpreted Java memory usage appears to approach a limit suggesting potential memory reuse and adequate garbage collection. On the other hand, Python continues to require more memory with more transmission iterations. One possible reason behind this result may be that Python continues to dynamically allocate memory as its applications transmit and receive data without reaching a point where garbage collection is deemed appropriate by the interpreter.

Ping Ring

A ring topology is simulated to examine the scalability of ns-3 when it must handle multiple DCE applications. The ring consists of a variable number of routers connected to one

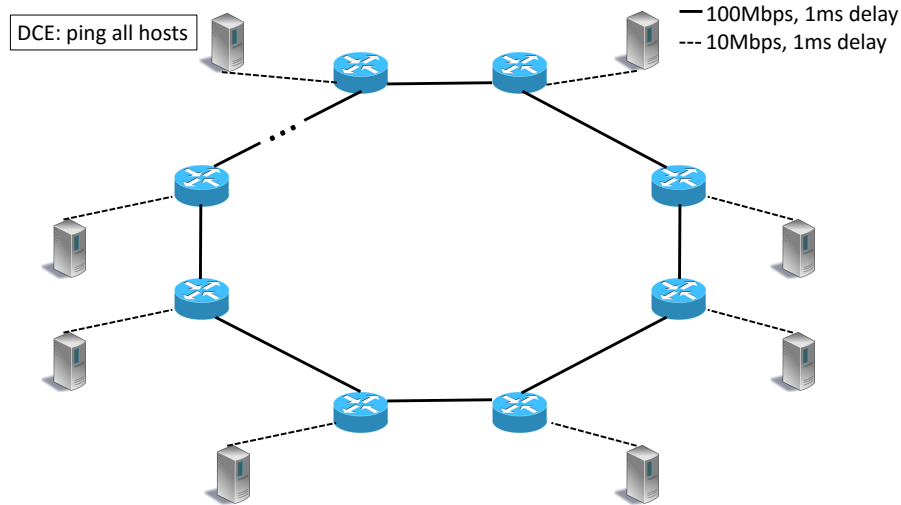


Figure 3.6: Ring topology

The ring topology is used to examine the scalability of ns-3 simulations when handling multiple DCE applications.

another in a circle as shown in Figure 3.6. Each router is connected to a single host. Network interfaces for all links are configured similarly to those in the dumbbell topology using different 255.255.255.0 subnets (CIDR /24) to again confirm packet transmission occurs through successful L3 routing. Within the topology, one end host is selected for DCE application installations. The end host attempts to reach every other end host in the topology. Pings are sent 1,000 times for each end host to inflate the amount of processing required for each DCE application.

Figures 3.7 and 3.8 display the results of the ring topology experiments. Starting multiple ping applications for the native code did not require significant overhead in terms of memory or time. Python produced wallclock times between Java and interpreted-only Java. However, its memory usage was significantly lower than the dumbbell topology results. The Java program benefited from the `-Xint` flag, producing lower wallclock times than the Python version with a slight memory usage improvement over the full JVM. However, both experimental runs of the Java program produced significantly higher resource usage than the other programs. This memory usage trend may have been a consequence of starting and stopping the JVM multiple times.

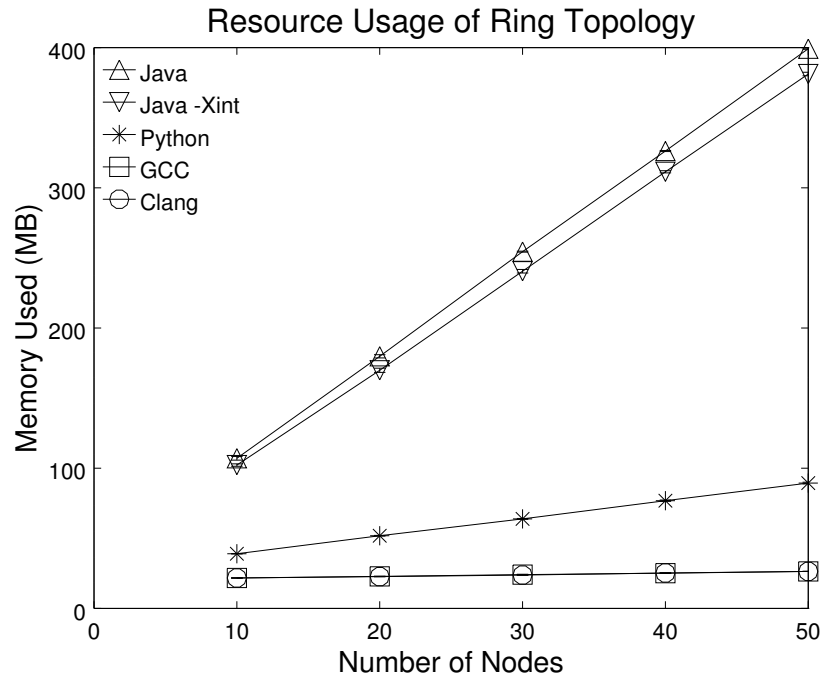


Figure 3.7: Ring topology memory usage
Ring topology results for total memory usage in MB. Standard error bars are displayed for each data point.

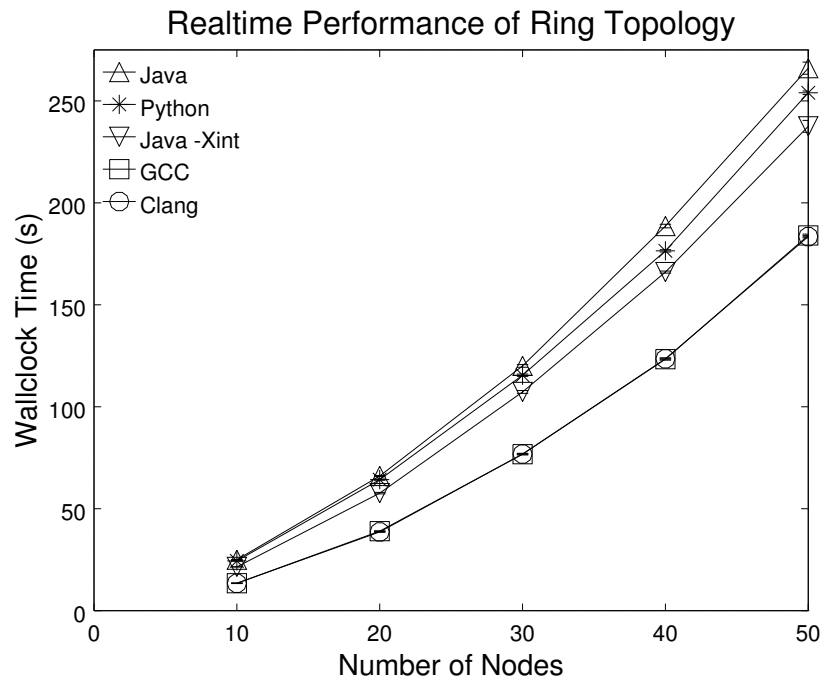


Figure 3.8: Ring topology wallclock execution time
Ring topology results for wallclock execution time. Standard error bars are displayed for each data point.

During testing of the OpenJDK within DCE, multiple applications running simultaneously within the limited address space of DCE made it difficult for the JVM to find free space for its code heap. Two Java applications running at the same time as in the dumbbell topology experiments appeared to be the stable limit while more than two simultaneous applications produced inconsistent successful results. DCE applications within the ring topology experiments were given staggered start times such that one would complete before another began. When Java applications had dedicated individual access to the Java runtime as in the ring topology experiments, the JVM resources could be successfully launched and deconstructed without interference. Testing for potential solutions to the simultaneous usage issue is ongoing. However, sufficient use cases are available for the current framework.

3.2 Enabling Simulation of GPU Network Applications

A framework has been designed in [54] that utilizes GPUs on a single system as shared resources so multiple nodes in an ns-3 topology can use the GPUs. Multiple options are available for providing this functionality. The API interacting with the GPU may be handled natively, simply allowing calls found in an application to directly interact with a GPU. Another option allows GPU usage to be virtualized in a similar manner to current sharing mechanisms in cloud computing. Multiple simulated nodes in the topology can access the GPUs in a transparent manner, as if they were directly connected to the resources. Regardless of design option, calls to the GPUs from the simulated applications require no modification to source code. Debugging and analysis of network applications employing GPUs can be administered in simulation, providing flexibility and scalability while reducing initial hardware requirements.

Two major design options are implemented, both in the context of *libcuda*, the CUDA driver library. One design looks at native integration of the API into the supported calls in the DCE POSIX layer. Virtualized integration is also designed and studied, employing the GPU virtualization service gVirtuS, whose backend communicates to a frontend built

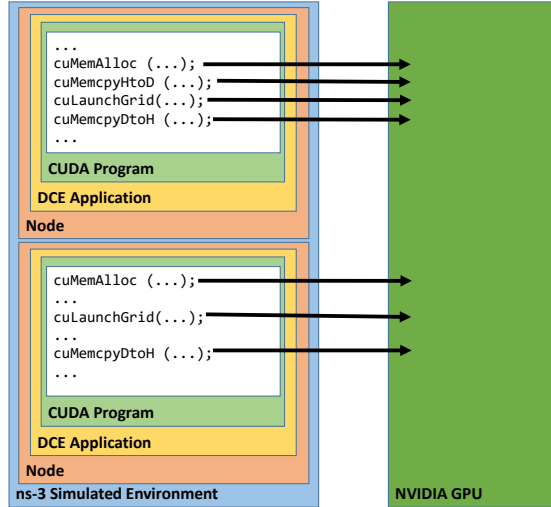


Figure 3.9: Designs for enabling CUDA support for DCE natively

into DCE. Communication between DCE and the backend occurs through either a simple file descriptor socket or a tap-bridged TCP/IP connection. Within ns-3 and DCE, the user interface is the same regardless of the underlying designs as shown in the identical *ns-3* blocks in Figures 3.9 and 3.10.

3.2.1 Native Integration

Native integration enabling CUDA support for DCE-enabled applications allows API calls to CUDA to direct to actual CUDA functions. This design can be seen in Figure 3.9 where CUDA calls in the given binary directly call to the GPU. In the DCE architecture, this integration is realized with the `NATIVE` macro. In this particular case, the functions are the CUDA driver API. These calls interact with the actual hardware installed on the system. Enabling native integration is straightforward. The design is simple and immediately provides the simulated framework.

3.2.2 Virtualized Integration

Virtualized integration in DCE enabling CUDA support employs the framework depicted in Figure 3.10. Under this design, the gVirtuS backend process acts as a server, listening for

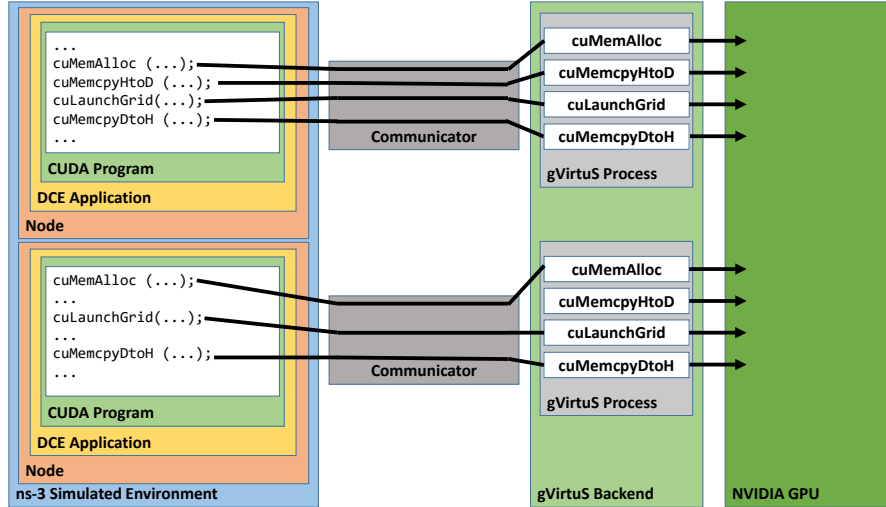


Figure 3.10: Design for enabling CUDA support for DCE with gVirtuS

incoming connections externally from the ns-3 simulator. When a DCE application in the ns-3 simulation encounters CUDA API calls in its installed binary, it connects to the back-end process. A subprocess is then created in the gVirtuS backend to maintain the context for the DCE application and its installed binary. CUDA API calls within the binary of that DCE application are channeled through a communicator associated with the connection. This communicator sends the API call and its associated arguments to the associated subprocess in the gVirtuS backend. This subprocess is then responsible for executing the actual CUDA API calls and interacting with the underlying NVIDIA GPU. If multiple nodes in the ns-3 simulation have CUDA-based binaries installed on DCE applications, each one establishes its own separate connection with its own communicator. Each communicator in turn connects to its own subprocess that is being handled in the gVirtuS backend. The design of this virtualized implementation accommodates introduction of simulated latency for particular API calls while native integration does not.

Communicators for the connections between DCE and the gVirtuS backend have been designed in two ways. One type of communicator handles a Unix-based file descriptor socket that sends and receives messages between the DCE applications and the gVirtuS backend subprocesses. Each communicator handles its own temporary file so that different

communicators are not be reading or writing to the same file. The other type of communicator allows DCE and the gVirtuS backend to communicate through tap bridges that act as network interfaces between the two processes. When this communicator is used, the gVirtuS backend proactively creates tap bridges to maintain distinct connections between each connected DCE application and gVirtuS backend subprocess. Both communicators are studied to compare their overheads.

3.2.3 Experiments

Single Node Benchmarks

A single node topology is not very useful for typical network research, but it provides a baseline for initial analysis of the system designs. A single node is simulated with one GPU-based network application installed. The following applications are studied:

- *Vector addition* (VECADD): Performs element-by-element vector addition.
- *Matrix multiplication* (MATMUL): Implements a matrix multiplication kernel.
- *Simple Texture* (SMPTEX): Performs a demonstration of CUDA textures.
- *Device Query* (DEVQRY): Displays the properties of the CUDA devices present.

The benchmark results are shown in Figures 3.11 and 3.12. The designs interfacing with the gVirtuS backend provide comparable wallclock execution times to the natively integrated implementation. The gVirtuS framework utilizing basic file sockets provided generally better execution times, suggesting it requires less network overhead than the tap interface. Resource usage was similar across the different designs, but the gVirtuS designs did exhibit some overhead from the gVirtuS backend processes.

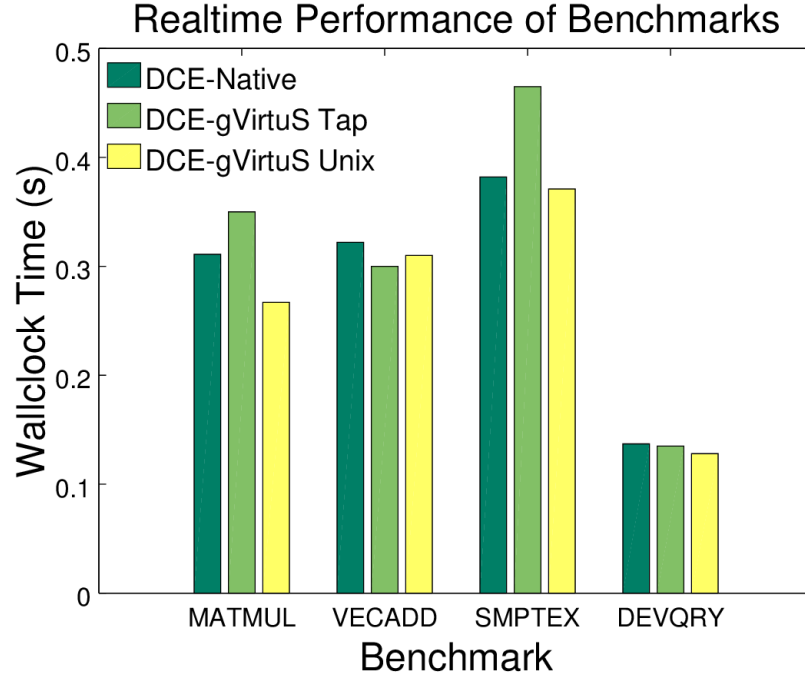


Figure 3.11: Single-node wallclock execution time
Benchmark results for wallclock execution time for a DCE-enabled node. Standard time errors are less than 8% of their reported values.

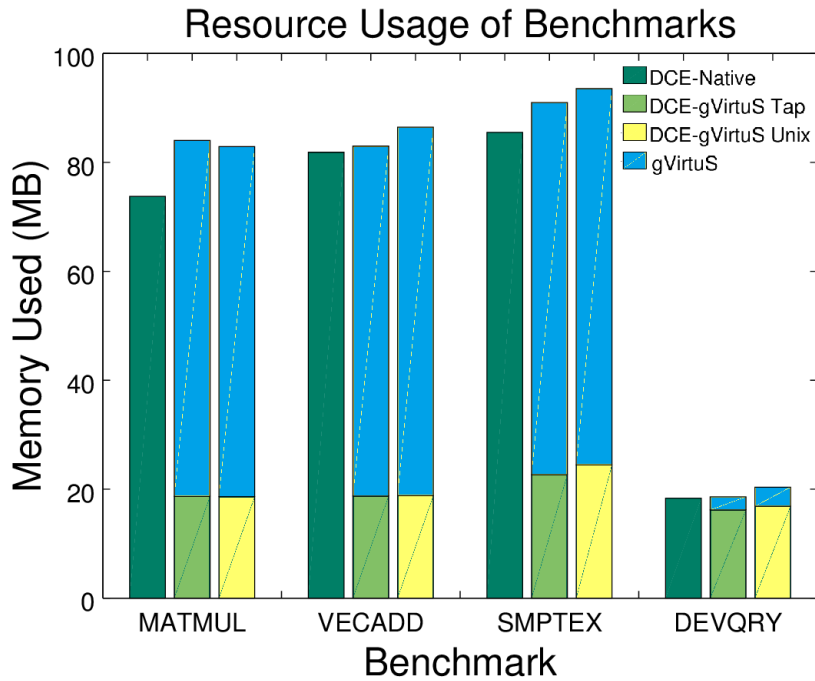


Figure 3.12: Single-node CPU memory usage
Benchmark results for total CPU memory usage for a single DCE-enabled node for each framework. Standard memory errors are less than 3% of their reported values.

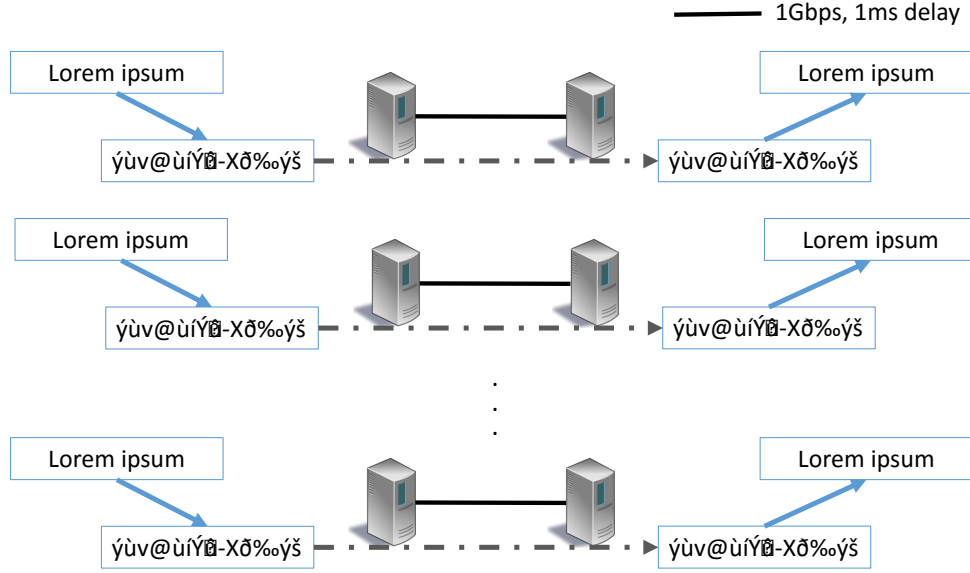


Figure 3.13: Pairs network topology

The “pairs” network topology is a disjoint set of node pairs where one node encrypts and sends data to the other node which decrypts it.

Pairs Topology

The “pairs” topology is a simple example to examine the scalability of the described designs using a practical GPU network application across multiple simulated nodes. As shown in Figure 3.13, pairs of nodes are only connected to each other. This topology reduces the overhead that would have occurred in a more connected topology. The major overhead that is visualized stems from the DCE applications and the underlying CUDA frameworks. The node pairs for this topology are enabled for DCE with CUDA-based AES encryption/decryption applications. The AES application encrypts and decrypts using the Electronic Codebook (ECB) mode. This mode is not secure for actual cryptographic protocols, but it presents the types of simulations enabled by this work. The sender nodes encrypt a 30MB file with the parallelized AES ECB encryption and send the encrypted data to the receiver nodes. Upon receiving the data, the receivers decrypt the data with parallelized AES decryption and write it to a file. Each node pair performs this process with a unique key.

The results for the pairs topology are shown in Figures 3.14, 3.15, and 3.16. Wallclock

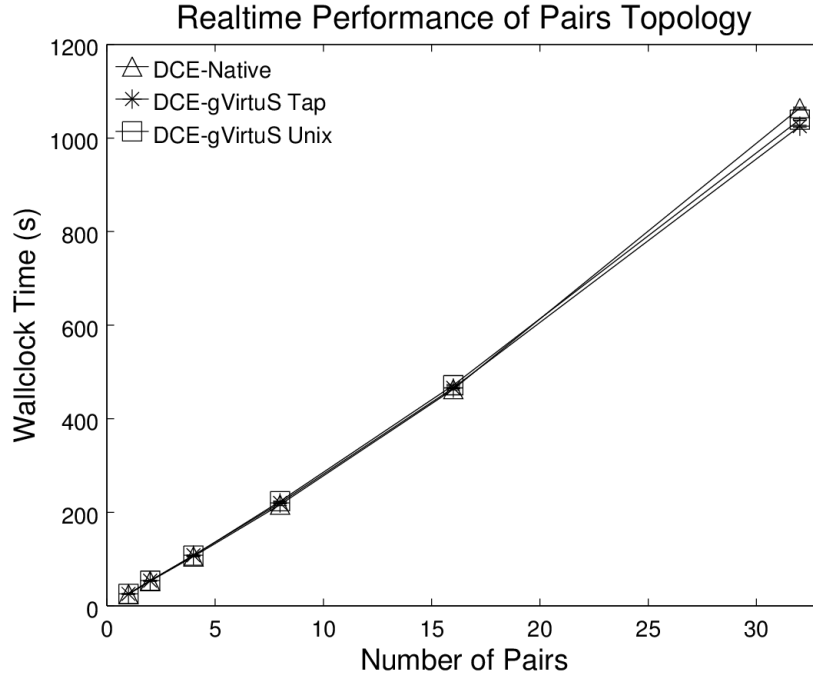


Figure 3.14: Pair wallclock execution time

Wallclock execution time results for the pairs topology executing AES encryption and decryption for each framework configuration. Standard errors are noted to be minimal.

execution time is nearly identical across all examined frameworks. Since each simulation is running the same kernel functions across the different nodes, the similar GPU memory results are expected. The CPU memory usage results are also similar across each framework. It should be noted that the CPU memory usage for the virtualized integration frameworks comprises resource usage from the main simulation process and each gVirtuS backend subprocess. Virtualized integration is preferred over native integration for future modeling improvements, and its minimal overhead validates this preference.

Further testing sought to determine the maximum number of nodes that could be simulated in the pairs topology. Originally, 64 simulated nodes are tested sharing a single GPU. The true limit in this case is 76 nodes (38 pairs) limited by the 6GB memory of the tested GPU. Sharing a single GPU over 76 simulated nodes is preferable to acquiring 76 GPUs. Also, multiple GPUs could be installed to allow more simulated nodes to share the combined resources on the underlying system.

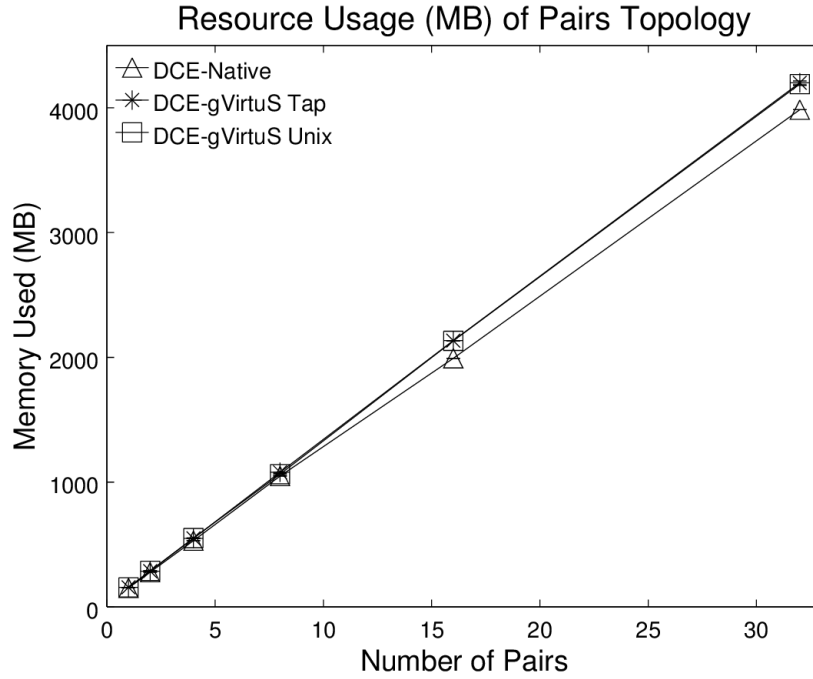


Figure 3.15: Pairs CPU memory usage

Total CPU memory usage results for the pairs topology executing AES encryption and decryption for each framework configuration. Standard errors are noted to be minimal.

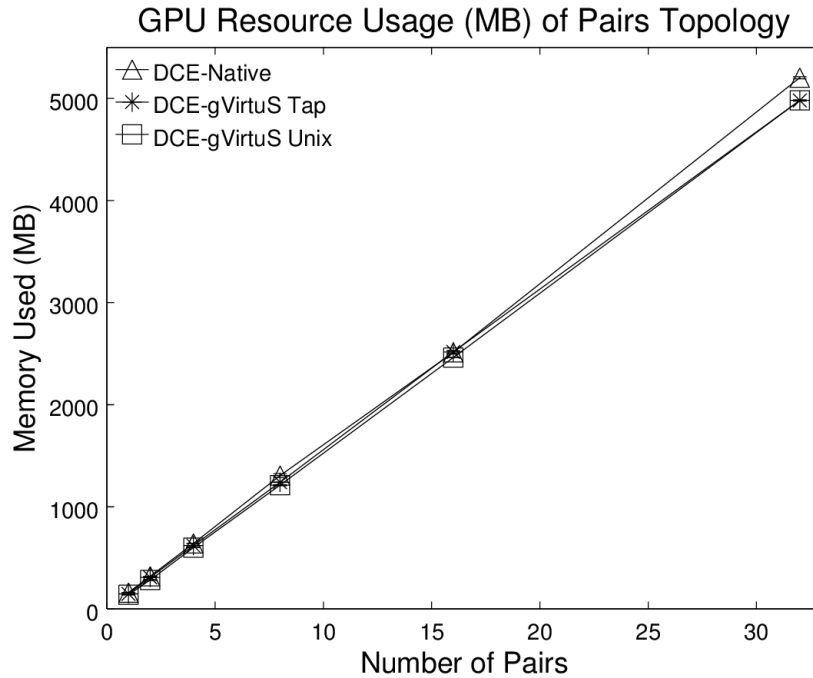


Figure 3.16: Pairs GPU memory usage

Total GPU memory usage results for the pairs topology executing AES encryption and decryption for each framework configuration. Standard errors are noted to be minimal.

CHAPTER 4

ANALYSIS OF EXISTING SDN SIMULATORS AND EMULATORS

An SDN simulation framework within ns-3 is introduced in [55] and extended in [56] employing DCE in conjunction with Python and the Python-based controller libraries POX and Ryu and an ns-3 user-defined application, `SdnSwitch`, that supports OpenFlow 1.0. Separate, subsequent work providing OpenFlow 1.3 switch support has been made to cooperate with DCE as well to permit SDN simulation using a more widely used version of the OpenFlow specification. This framework applies a novel approach toward achieving scalable, portable simulation of SDN-based topologies. Providing a mechanism for using POX or Ryu within DCE, controller applications written for these libraries can be developed and debugged in simulation and then immediately ported to a real-world deployment. Benefiting from the current capabilities provided by both ns-3 and DCE, this framework can achieve significant simulated node scales, providing the capacity necessary for adequately simulating enterprise and data center networks. A number of SDN simulation/emulation libraries are also examined in addition to the previously described framework. Comparing real-time performance, memory usage, and reliability in terms of packet loss, the overall performance of each simulation/emulation tool can be compared against one another. Through this comparison, appropriate tool selection can be determined.

4.1 SDN Simulation with ns-3 and DCE

OpenNet presents a performance bottleneck in terms of scalability due to its TAP device interface between Mininet and ns-3. Mininet suffers from scalability issues as well due to its heavy use of network and process resources. The SDN capability in baseline ns-3 exhibits its own set of issues as previously described in Section 2.3. For these reasons, an SDN switch application supporting OpenFlow 1.0 has been designed to interface with

the DCE module of ns-3 to allow real, deployable controller applications to be executed on an ns-3 simulated topology. A separate, subsequent work independent from this effort provided OpenFlow 1.3 switch support in ns-3 as well. The source code from that work has been made to cooperate with DCE in this work to provide similar functionality to the OpenFlow 1.0 work while supporting a more robust and widely used version of the OpenFlow specification.

The design of the classes specific to providing SDN simulation capabilities in ns-3 primarily center on implementing an OpenFlow-enabled switch as a user-defined application. This application is installed on nodes in the simulated topology, allowing them to receive packets, perform a given set of actions based on the nature of these received packets, and then forward them appropriately. The switch is designed such that it can communicate with external, real-world controller libraries. The SDN switch application is comprised of the `SdnPort`, `SdnFlowTable`, and `SdnSwitch` classes. `SdnPort` provides the formal definition of a binding port for the switch to send and receive data. The `SdnFlowTable` provides the structure and control for a table of flow rules for the switch to use on incoming packets. The `SdnSwitch` provides the actual application acting as a switch.

The `OFSwitch13` module[57] enhances ns-3 with OpenFlow 1.3 technology support. The module components include the switch network device, the controller application interface, the OpenFlow channel, and the external *ofsoftswitch13* library. The switch network device is used to interconnect ns-3 nodes using CSMA devices and channels. Each switch device consists of a collection of ports, and each of the ports is connected to a CSMA device. The switch is connected to a controller either through a CSMA channel shared by all controlled switches or its own dedicated CSMA or point-to-point channel. Each switch receives packets from one port, directs them to the *ofsoftswitch13* library[58, 59] for OpenFlow pipeline processing, and then executes the appropriate actions based on the returned action set. It estimates the average flow table search time by introducing a $k * \log_2(n)$ delay derived from the concept that most lookup algorithms are based on binary search trees. One

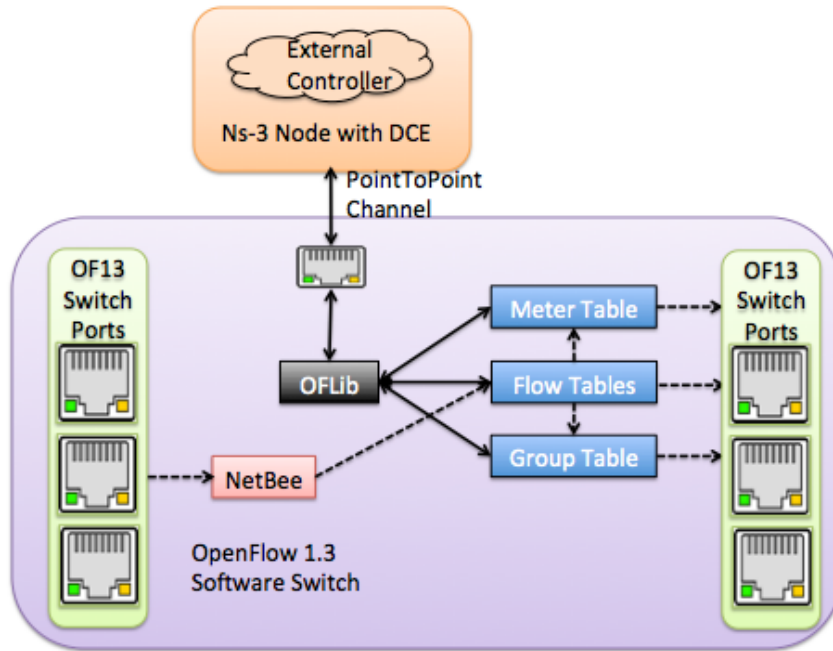


Figure 4.1: Overall architecture of OFSwitch13 connecting to a DCE-enabled node.

or more OpenFlow queues can be attached to a port to provide QoS support for the output packets. An OpenFlow 1.3 controller interface and an OpenFlow channel also accompany the module to provide basic functionality for controller implementation. However, instead of using this non-portable interface, this work examines the use of DCE to integrate external OpenFlow controllers, such as POX and Ryu. The overall architecture is shown in Figure 4.1.

The *ofsoftswitch13* library provides the OpenFlow datapath implementation for OFSwitch13, including the input/output ports and flow, group, and meter tables. It operates entirely in user space and uses the OFLib library to convert internal messages to and from OpenFlow 1.3 format. The packet-processing NetBee[60] library is used to decode and parse incoming packets. The library is modified to integrate with the OFSwitch13 module. In order to send and receive packets to and from the ns-3 environment directly, the related functions are annotated as weak symbols to permit overriding them at link time. A similar strategy is applied for time-related functions to ensure time consistency between the library

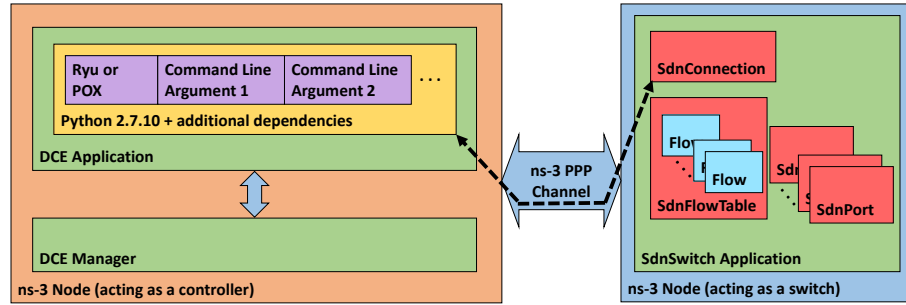


Figure 4.2: Structure of a DCE-enabled node as a controller and an `SdnSwitch` object. Communication occurs across the ns-3 point-to-point channel object. Packets are handled by the `SdnSwitch` through ns-3 simulated sockets maintained in the `SdnConnection` object. DCE handles packet coordination on the controller node.

and simulator. The library uses callbacks to notify the module about internal packet events.

Using both ns-3 and DCE, the simulated switches can use either the `SdnSwitch` class for OpenFlow 1.0 support or the `OFSwitch13` objects for 1.3 capabilities. The simulated controllers can take DCE applications running either POX or Ryu. In this way, the controller applications examined are directly portable to real network deployments. Setting up the entire framework utilizes the installation tool *bake*[61].

Figure 4.2 displays how a POX or Ryu application on a DCE-enabled node would interact with the `SdnSwitch` object. The controller is managed by DCE. The DCE Manager maintains the simulated operating system space for the node and interacts within the node with the DCE Application. The installed application on this DCE Application is a locally built version of Python 2.7 with the appropriate library dependencies for POX and Ryu. Python interprets the appropriate instructions, executes the specified controller applications, and communicates packets back and forth to the ns-3 channel (*point-to-point usage shown*). Socket calls for packet transmission and reception by Python are translated to simulated socket calls for ns-3 by DCE. Listing 4.1 displays the code for configuring and installing POX or Ryu on a DCE-enabled node. Python is set as the binary to execute, and environment variables are set to point the executed process to appropriate locations in the simulation filesystem for Python scripts. The managing script for the controllers and their

specific applications then simply become arguments to be read by the Python binary.

4.2 Experiments

Experiments have been performed on some of the emulation and simulation frameworks described in Section 2.3. These experiments and the topologies on which they have been executed are not identical across frameworks due to differences in design and supplied capabilities as well as various limitations that are noted as necessary. Even so, the experiments have been designed in such a way to make them as similar in terms of topology, network configuration, and traffic generation as possible. As an additional consideration for the flow-based simulator *fs*, two definitions for the size of a flow are examined. A flow may model either 10 packets (denoted as “*fs-sdn 10*” in the figures) or 100 packets (denoted as “*fs-sdn 100*”).

Determining the resource utilization of each of the simulation/emulation tools varies based on the nature of the tool. Because *fs-sdn* and the ns-3 DCE framework are both simulators and handle all of their respective components internally, their memory usage can be considered the maximum usage achieved by the process under which they reside. For Mininet and OpenNet, it is not quite as simple. As emulators rather than simulators, these tools employ a number of components external to the process on which they reside. It is important to gather as many of these external components as possible. Each Mininet host, switch, and controller requires its own user space and memory. For each instantiation of traffic generation (*iperf*, *ping*, etc.), a process is spawned as well, and its information must be considered as part of the Mininet resource profile. Furthermore, for each controller required for a particular topology, its process data must be collected since it effectively participates in the emulation. For the largest campus network topology examined, 21 controller processes are examined as part of the Mininet and OpenNet emulations. Additionally, Mininet spawns *dhclient* processes to dynamically allocate IP addresses for all of the objects that it creates. However, OpenNet utilizes its ns-3 components for determining IP

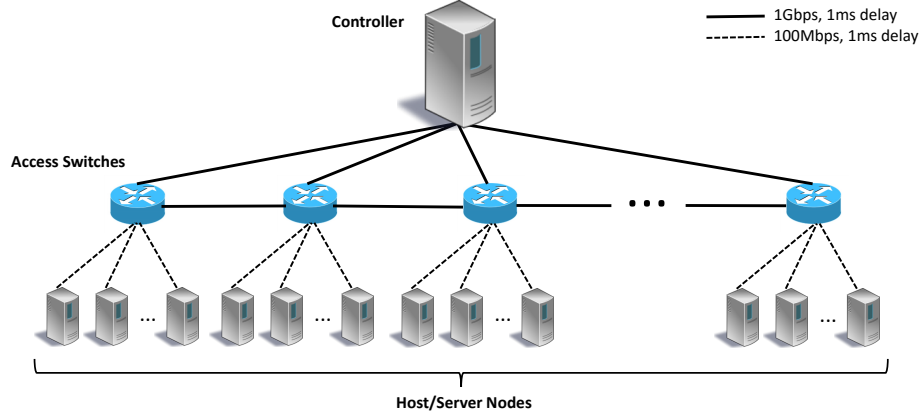


Figure 4.3: Linear network topology

The linear network topology is a linear network of 10, 20, 30, 40, and 50 access layer switches each connected to 16 hosts.

addresses so it does not require this extra process consideration.

The linear network topology, as shown in Figure 4.3, is designed as a linear network of switches. This single layer comprises a set of switches that connects to the hosts of the network. These access switches connect to each other as well as the controller. For traffic generation, each host randomly selects another access switch and then sends its data to its corresponding host on that switch, i.e. host 2 on switch 0 might send to host 2 on switch 8. A single POX/Ryu controller is connected to each switch in the topology. This controller forwards traffic through layer 2 learning. The controller directs switches individually to flood new traffic while recording the input port and source Ethernet address. When the switch receives another packet destined for a recorded Ethernet address, it forwards it through the associated switch port.

For the simple linear switch topology, performance in terms of wallclock completion time, shown in Figure 4.4, does not differ too significantly for Mininet running both POX and Ryu, *fs-sdn* with 10-packet flows, and the ns-3 DCE framework running POX. However, the best performance is exhibited by *fs-sdn* with 100-packet flows. This result can be expected since the lower level of granularity at which *fs-sdn* is operating will cause fewer events to be created for processing. Furthermore, the flow granularity can be assumed to be at least part of the reason for the resource hierarchy between 100-packet flow *fs-sdn*,

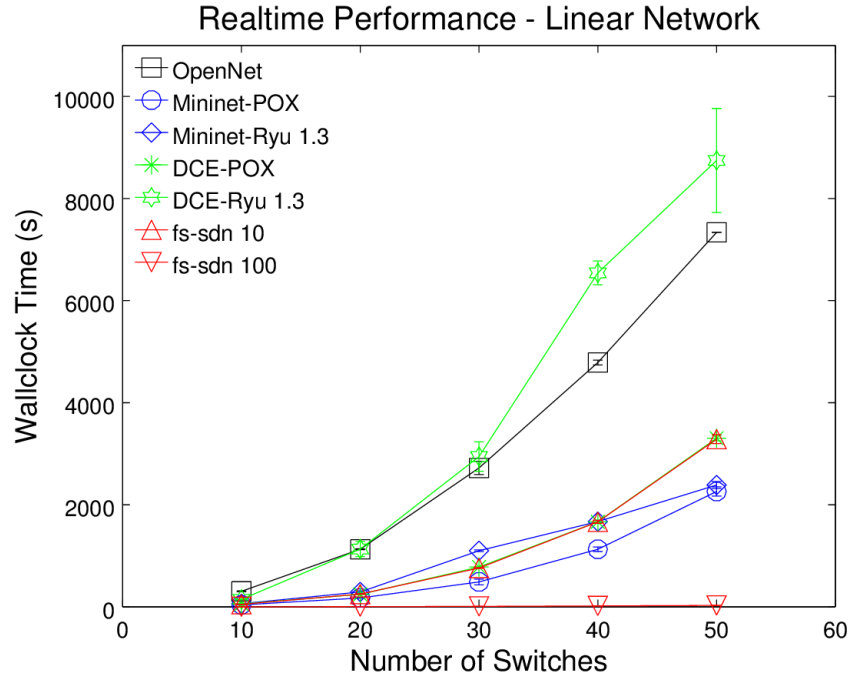


Figure 4.4: Linear network real-time performance
Linear network topology results for real-time performance in seconds. Standard error bars are displayed for each data point.

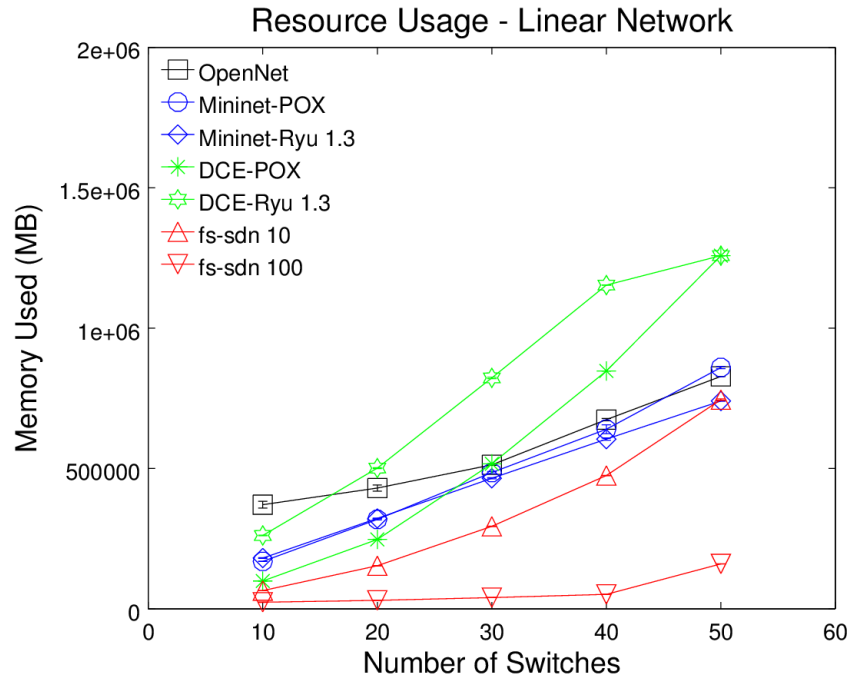


Figure 4.5: Linear network resource usage
Linear network topology results for resource usage in MB. Standard error bars are noted as negligible.

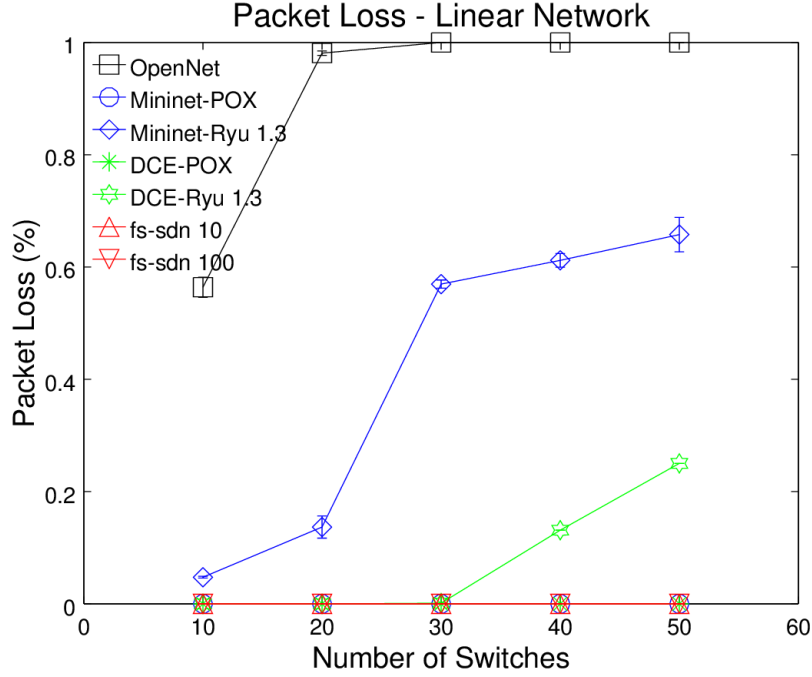


Figure 4.6: Linear network packet loss

Linear network topology results for percent packet loss. Standard error bars are noted as negligible.

10-packet flow *fs-sdn*, and the ns-3 DCE framework displayed in Figure 4.5. Interestingly, DCE running Ryu and interacting with the OFSwitch13 objects exhibited the highest real-time overhead and some of the highest resource usage. Since POX and Ryu are utilizing the same type of application to implement their packet forwarding behavior, much of this overhead is most likely a consequence both of the underlying Ryu architecture and the OFSwitch13 module. Although little difference is noticed between POX and Ryu in Mininet outside of packet loss, the simulations running in ns-3 will encounter processing overhead as DCE balances the threading mechanisms of Ryu and OFSwitch13 interfaces with the *of-softswitch13* library for pipeline processing. Shown in Figure 4.6, in terms of packet loss, Mininet experienced significant loss when handling OpenFlow 1.3 in relation to Ryu that is not realized when it runs OpenFlow 1.0 and POX. Similar but less significant packet loss is observed for DCE when running OFSwitch13 and Ryu at higher scales, which may suggest that this type of loss is to be expected. Furthermore, the lower packet loss rate exhibited

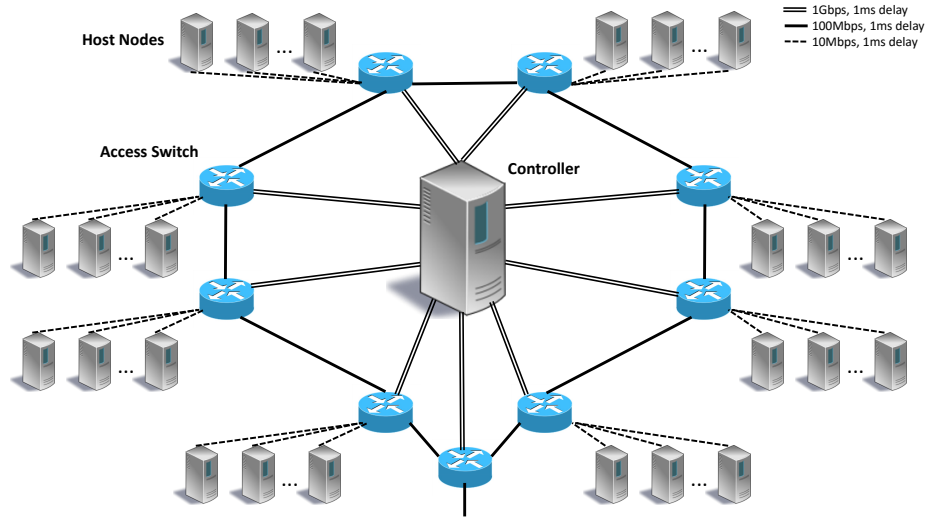


Figure 4.7: Simplified campus network topology

The campus network topology connects a ring of switches modeling a simplified campus network.

by DCE while running Ryu can partially explain the higher runtime realized compared to Mininet. Under the same circumstances, the ns-3 network simulation will be processing a greater amount of successful packet transmissions.

The campus network topology examines a ring of simplified campus networks as shown in Figures 4.7 and 4.8. Each campus network is simply a ring of switches that each connect to their own set of 4 hosts. Eight switches in the ring are considered *access* switches. An additional switch in the ring is considered a *gateway* switch that connects to a single *exchange* switch. The exchange switches then form a ring themselves to connect all of the campus networks. Each ring, including the exchange ring is controlled independently by a POX controller. These controllers execute link layer discovery to determine the part of the topology they control and a spanning tree application that prevents flooding loops from occurring. The controllers forward traffic with layer 2 learning. For traffic generation, each host decides whether to send data on its local ring or to a remote ring. It randomly selects the switch and receiver.

Due to the design limitations of *fs-sdn*, a minor modification to the described topology has been conceded for its experiments. Since *fs-sdn* only operates at layer 3 and above, link

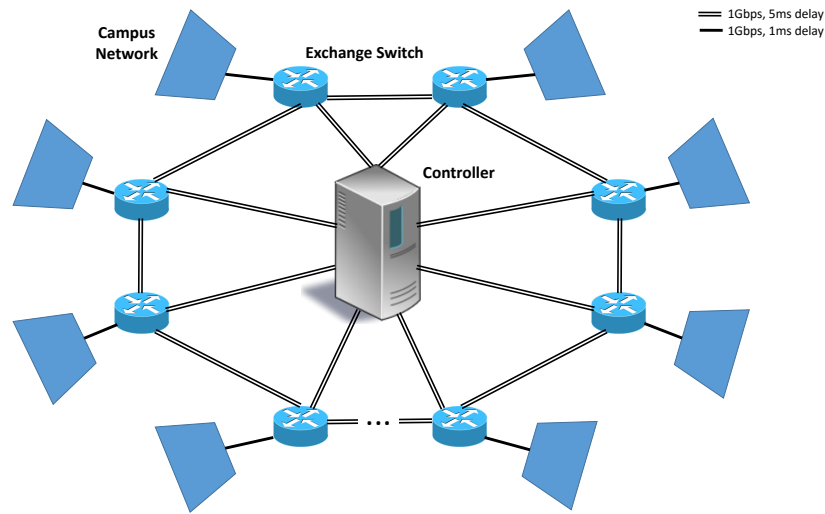


Figure 4.8: Ring of campus networks

The campus network topology connects the simplified campus networks together, forming a ring of these smaller networks.

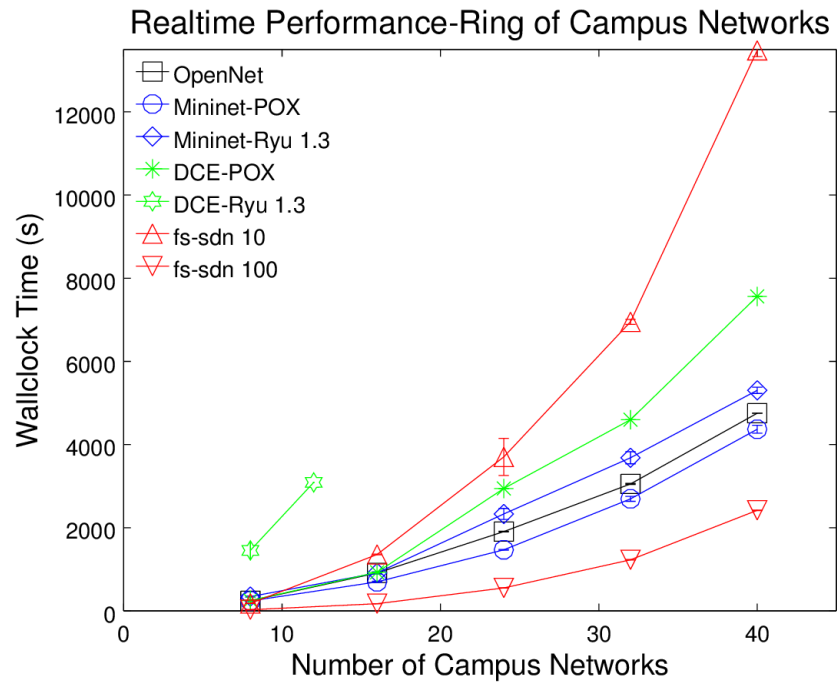


Figure 4.9: Campus network real-time performance

Campus network topology results for real-time performance in seconds. Standard error bars are noted as negligible.

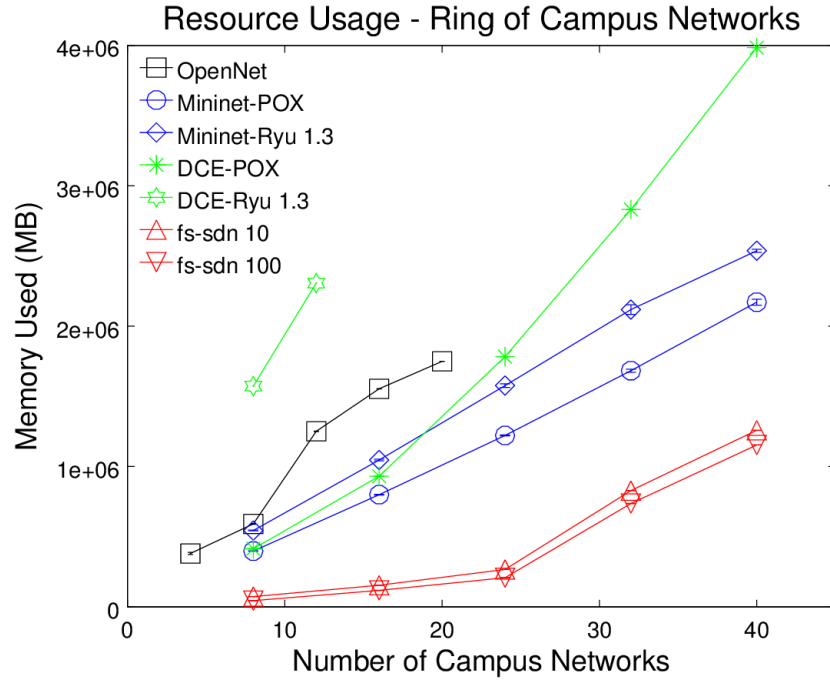


Figure 4.10: Campus network resource usage
Campus network topology results for resource usage in MB. Standard error bars are noted as negligible.

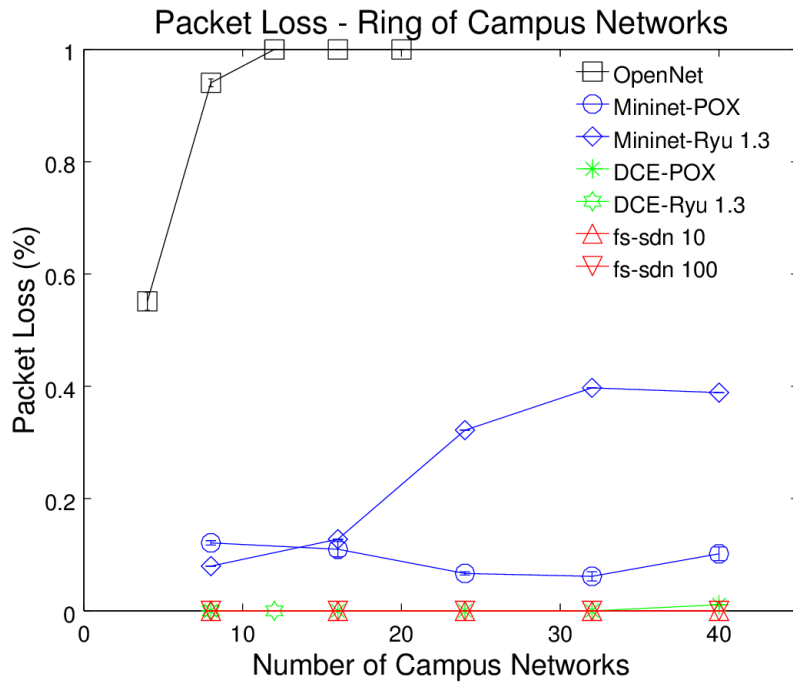


Figure 4.11: Campus network packet loss
Campus network topology results for packet loss percentage. Standard error bars are noted as negligible.

layer discovery and spanning tree controller applications cannot prevent flooding loops. To accommodate this limitation, one link from each ring in the network is deleted. In these simple ring topologies, this modification produces an effect similar to flooding loop prevention while maintaining roughly the same network behavior. The controller applications are all still allowed to execute as well to ensure as much network traffic is maintained as possible. For this specific topology, it is adequate to administer this change for the experiments on *fs-sdn* while still comparing its results against the other emulators/simulators. It would not be appropriate for more complex topologies as the change to network traffic would be drastic.

For the campus network topology, shown in Figures 4.9, 4.10, and 4.11, 100-packet flow *fs-sdn* again outperforms the other systems in terms of execution time and memory usage. However, 10-packet flow *fs-sdn* does not perform as well as Mininet or the ns-3 DCE framework. Resource usage in both Mininet and OpenNet demonstrate the overhead required by these two systems as they create greater numbers of *veth* pairs and occupy more process space with additional hosts. For OpenNet, scaled testing is forgone as its results continue to be unreliable. Again, at higher scales, the ns-3 DCE framework running OpenFlow 1.0 and POX produces better performance times than Mininet and 10-packet flow *fs-sdn*, but scaling in terms of resource usage looks to become an increasing issue. However, the memory requirements for the ns-3 DCE framework, at approximately 4GB, remain in the comfortable realm of possibility for current computer systems. An interesting and significant limitation of running the ns-3 DCE framework with OpenFlow 1.3 and Ryu is its current inability to scale beyond 12 campuses, as observed by the lack of data points reported in Figures 4.9, 4.10, and 4.11. A significant debugging effort has been conducted, but this limitation appears to be the result of simulated contextual limitations in DCE as they relate to Ryu importing greater numbers of dependencies within its scripts. As a best effort, the maximum capability provided through OpenFlow 1.3 and Ryu is demonstrated, but efforts are ongoing to resolve this issue.

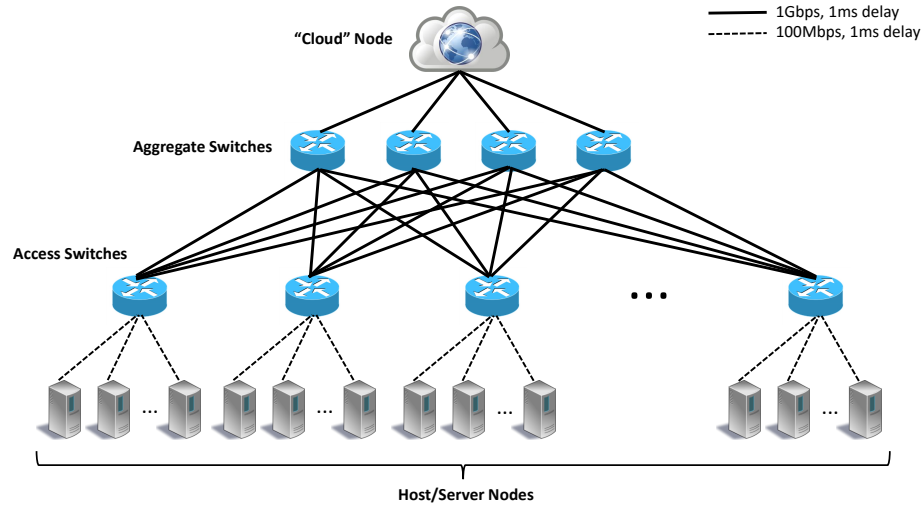


Figure 4.12: Data center network topology

The data center network topology is a leaf-spine with 4 aggregate switches fully connected to 4, 8, 12, 16, or 20 access layer switches. Not shown: All switches connect to a single controller with 1Gbps channels with 1ms delays.

The data center network topology, as shown in Figure 4.12, is designed as a simple *leaf-spine* architecture composed of two layers of switches. One layer, referred to as the access switches, forms the leaf switches that connect to the hosts or servers of the data center. These access switches are also fully connected to a second layer of aggregate switches comprising the spine of the network. As a simplification, the aggregate switches connect to a single node, referred to as the “cloud” node. This “cloud” node models the gateway for the data center to either other data centers or the broader internet. The number of host/server nodes connected to each access layer switch is fixed at 8. Each host decides with equal probability whether to send data to the “cloud” or to a host on another switch. If the host does not send to the “cloud” node, it randomly selects an access switch and sends its data to its corresponding host on that switch similarly to the linear network experiments. A single controller is connected to each switch in the topology. This controller executes link layer discovery to determine the topology and a spanning tree application that prevents flooding loops from occurring. The controller forwards traffic through layer 2 learning. Limitations in the design of *fs-sdn* prevent it from adequately performing network discovery and

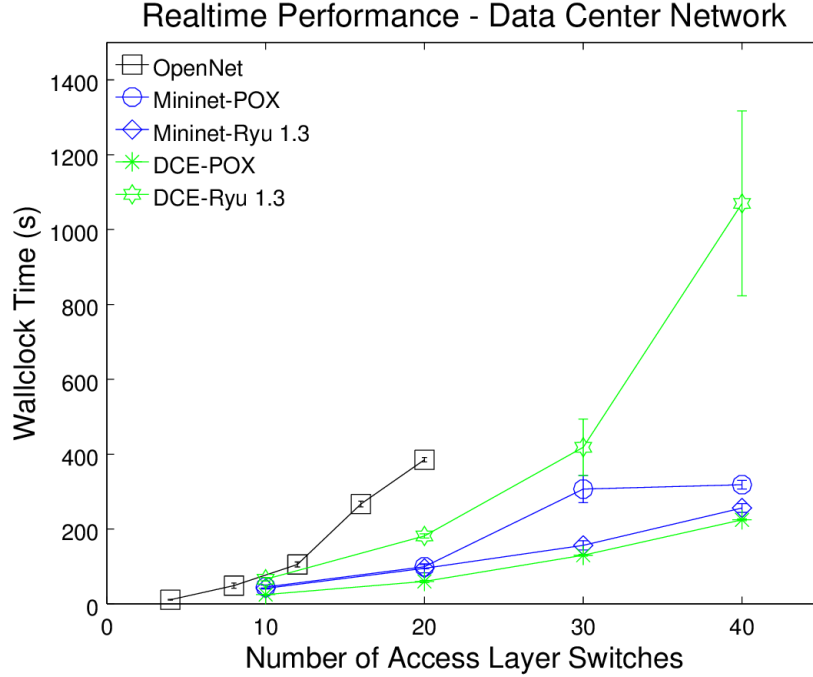


Figure 4.13: Data center network real-time performance

Datacenter network topology results for real-time performance in seconds. Standard error bars are displayed for each data point and noted as negligible except for DCE-Ryu points.

flooding loop prevention. Modifications to the actual topology would alter network traffic behaviors too significantly for a “fair” comparison, and as such, results are not gathered for the data center network topology using *fs-sdn*.

With results shown in Figures 4.13, 4.14, and 4.15, the datacenter network topology indirectly displays the design limitations of *fs-sdn* while presenting a case where the ns-3/DCE framework with OpenFlow 1.0 and POX can outperform Mininet and OpenNet in terms of real-time performance and memory usage. The data center itself is a more complex topology than the previous two, but its scaling is not tested quite as heavily. For OpenNet, scaled testing is forgone as its results continue to be unreliable. Packet loss for OpenNet, while not achieving 100%, is still non-zero and increasing with the topology size. This particular result helps to illustrate how TAP bridge resource capacity is approached with greater topology sizes. For OpenNet, the resource usage for the data center topology actually begins to decrease for topologies with more than 12 access layer switches. This

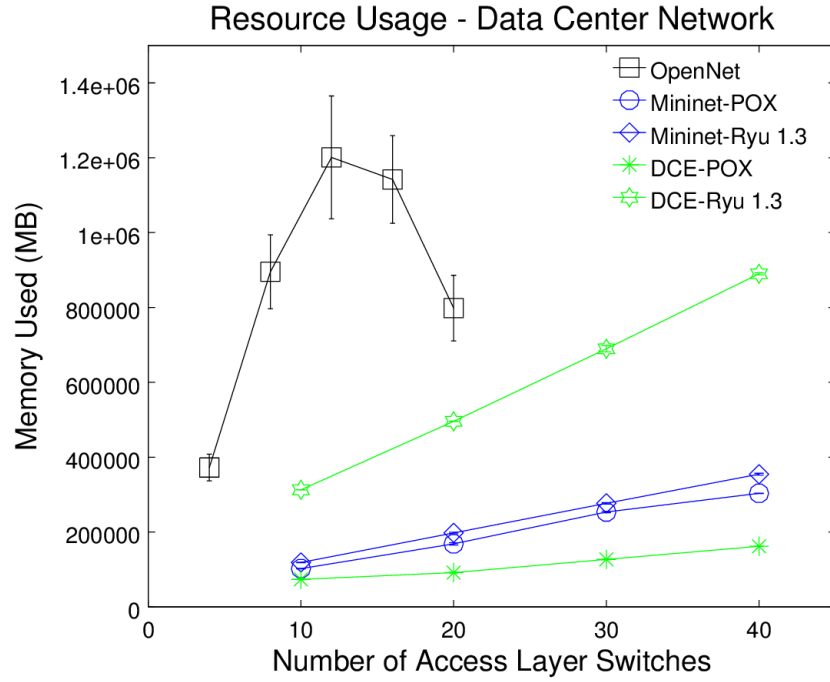


Figure 4.14: Datacenter network resource usage
Datacenter network topology results for resource usage in MB. Standard error bars are displayed for each data point and noted as negligible except for OpenNet.

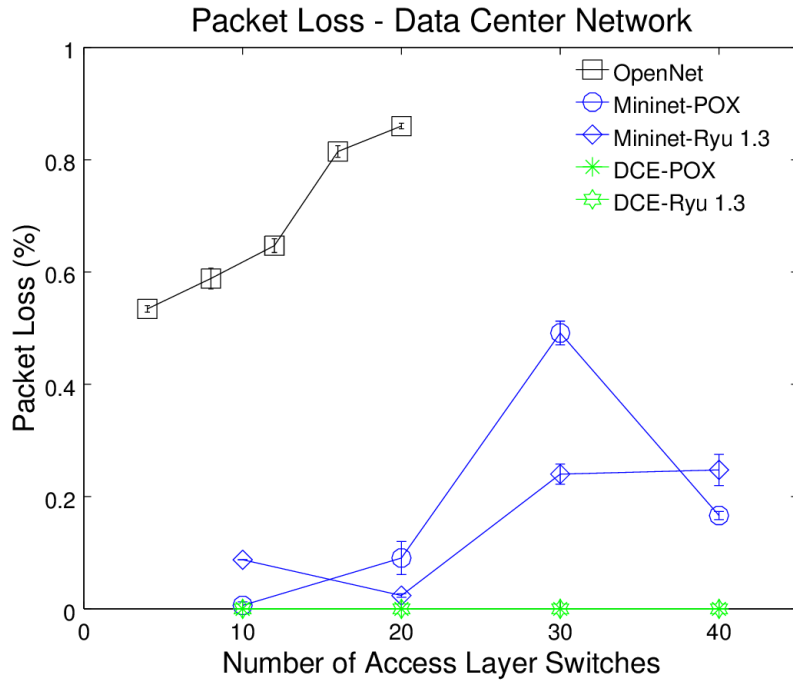


Figure 4.15: Datacenter network packet loss
Datacenter network topology results for packet loss percentage. Standard error bars are displayed for each data point and noted as negligible.

result suggests that as OpenNet nears the limits for TAP bridge allocation it uses less memory since fewer connections are successful. With fewer connections, the controller will not store as much network information and will not install as many flow rules, requiring less memory. The ns-3 DCE framework performs similarly to Mininet in terms of time when running OpenFlow 1.0 and POX but continues to exhibit higher wallclock times and memory usage with increasing scales. However, the ongoing reliability of the ns-3 DCE framework in terms of its low packet loss can both explain its higher performance requirements while demonstrating traffic fidelity that is missing from Mininet due to its increased virtual networking overhead.

Listing 4.1: Source code for installing POX or Ryu controller on a node in ns-3 using DCE.

```
1 DceManagerHelper dceManager;
2 DceApplicationHelper dce;
3 ApplicationContainer apps;
4
5 dceManager.Install(controllerNode);
6
7 dce.SetStackSize (1<<20);
8
9 // Python controllers
10 dce.SetBinary ("python2-dce");
11 dce.ResetArguments ();
12 dce.ResetEnvironment ();
13 dce.AddEnvironment ("PATH",
14     "/:/python2.7:/pox:/ryu");
15 dce.AddEnvironment ("PYTHONHOME",
16     "/:/python2.7:/pox:/ryu");
17 dce.AddEnvironment ("PYTHONPATH",
18     "/:/python2.7:/pox:/ryu");
19 if (controller == POX)
20 {
21     // POX arguments
22     dce.AddArgument("pox/.py");
23     dce.AddArgument("--unthreaded-sh");
24     dce.AddArgument("forwarding.l2_learning");
25 }
26 else if (controller == RYU)
27 {
28     dce.AddArgument("ryu-manager");
29     dce.AddArgument("ryu/app/simple_switch.py");
30 }
31 else
32 {
33     NS_LOG_ERROR ("Unsupported controller");
34 }
35 apps.Add(dce.Install(controllerNode));
36 apps.Start(Seconds (0.0));
```

CHAPTER 5

DEPLOYING SIMULATION ROUTING AS SDN APPLICATIONS

Similarly to simulation, SDN provides an environment where an entire topology is controlled collectively. As such, the mechanisms that are used to manage routing decisions in simulation can be leveraged for use in SDN. Applications using controller libraries such as Ryu or Floodlight can be used to discover information regarding the underlying controlled topology, calculate routes through the network, and install forwarding instructions on OpenFlow-enabled switches.

5.1 SDN-Based NIX Vector Routing

NIX vectors are an efficient method for computing network routes. As discussed in Section 2.5, they have been deployed in simulation and wireless routing to provide a stateless protocol that provides low latency and reduced memory requirements over traditional route computation. However, the rigid nature of IPv4 and slow adoption of IPv6 made its adoption in real wired networks impractical. With SDN, route determination is more flexible, allowing new methodologies to be introduced and evaluated. As such, NIX vectors can be designed into an SDN controller application in order to compute routes and install forwarding instructions on OpenFlow-enabled switches based on the calculated information.

NIX vectors, as originally proposed, are simply binary arrays that can be used by routers to determine through which port they should forward a packet. In addition to this bit array, the maximum length of a NIX vector L_{max} and its current actual length at a particular hop (L_h) are transmitted along with a packet. Each router within the network knows as a constant the total number of ports P_h through which it could send data, and this value determines how many bits must be stored for each particular hop in a route, namely $C_h = \lceil \log_2(P_h) \rceil$. In creating the NIX vector, a particular router can determine the ap-

appropriate output port for a packet based on traditional networking methods, and its binary representation is copied onto the C_h bits allocated for this particular hop in the route. The actual length of the NIX vector is then updated such that the next hop in the path knows where to place its output port data, i.e. $L_{h+1} = L_h + C_h$. Subsequent packets in this traffic flow can then use the created NIX vector rather than a traditional routing table, saving both memory space and lookup time. In reading a created NIX vector, a router on the path retrieves L_h to determine where to begin reading it. It then reads off C_h bits to determine its forwarding decision. It subsequently updates L_{h+1} to allow the next hop to know where to begin reading the NIX vector.

Translating the traditionally proposed NIX vector mechanism to one based on SDN principles requires some minor considerations. Through topology discovery (generally LLDP or BPDU), the controller knows the switches and links that it is tasked to manage. However, it does not immediately know which hosts or external networks exist outside of the network it controls. Through ARP flooding either on the edges or throughout the network (assuming spanning tree or some other loop prevention is in place), this missing picture of the source and destination just outside the edge of the controlled network can be ascertained. This design consideration also presents a requirements limitation in the extent that SDN-based NIX vector routing is initially constrained to a topology under a single (logical) controller.

The most complete transformation of the NIX vector concept to an SDN solution requires the addition of a NIX vector header that can be understood by both the controller library and the switches deployed on the controlled network. When a packet from a known host, i.e. a known Ethernet or IP address, is received at the edge of the network and is destined for a known recipient, the controller would determine the appropriate route to take through any number of path finding algorithms. It would then allocate a NIX vector based on the total number of ports at each switch along the route and copy the binary representation of the appropriate output port at the respective locations in the NIX vector. It can

instruct the edge switch to push this NIX vector onto the packet and forward it out of the appropriate port. Subsequent switches in the route would have flow rules preemptively installed that would match on a NIX vector Ethertype to know to read off the appropriate bits based on the current NIX vector length and the number of ports at that switch. It can update the NIX vector length for the next hop and forward the packet based on the bits that it read. As previously mentioned though, this design requires significant modifications to both the network controller library and switch source code in order to accommodate a new packet header and its associated rules.

Alternatively, an immediately deployable solution can exploit currently existing headers and OpenFlow rules to provide a routing solution influenced by NIX vectors. A common criticism of NIX vector routing is its resemblance to Multiprotocol Label Switching (MPLS), although the two concepts are distinct. Even so, the components that accommodate MPLS within the OpenFlow specification can be leveraged to prototype NIX vector behavior within SDN through a process that can be referred to as NIX-MPLS. The first 20 bits of the 32-bit MPLS header refer to the MPLS label which traditional routers use to distinguish specific traffic flows and swap based on predetermined traffic engineering principles. A single *bottom-of-stack (BOS)* bit is also contained in the MPLS header to denote whether an MPLS header is the “last” one, signifying that no additional MPLS headers will exist on the packet following the decision based on its label.

The introduction of OpenFlow version 1.1[62] and subsequent specifications delivered the opportunity to push and pop MPLS headers using an OpenFlow-enabled switch. These actions correspond to the `OFPAT_PUSH_MPLS` and `OFPAT_POP_MPLS` action types, respectively. These concepts provide an avenue for deploying NIX-MPLS behavior. The MPLS label can act as a representative space for determining an output port decision, matching only on this field rather than any additional fields such as Ethernet or IP addresses. In this way, the NIX-MPLS behavior exhibits the following workflow. Upon completion of the controller-to-switch handshake, switches on the network are given two flow rules for

each port that they contain. Each rule matches on the MPLS unicast EtherType (0x8847) and an MPLS label corresponding to a particular port. Each flow rule also matches on the BOS bit of the MPLS header. For MPLS labels that do not have the BOS bit set, the outer MPLS header is popped from the packet, and the packet is forwarded out of the port matching the popped MPLS label. The EtherType remains 0x8847. If the BOS bit is set, the outermost MPLS label is popped from the packet, but the EtherType is reset to IPv4 (0x0800). The packet can then be forwarded out of the port matching the popped MPLS label to presumably exit the edge of the controlled network. (For additional security, a more complex association of MPLS label and output port may be utilized, either through a hash function or some other obfuscation.)

Creation of the original “strand” of MPLS headers is instantiated at the incoming edge switch. When a new IPv4 packet is received by an edge switch, it forwards information to the controller. The controller uses the received packet information, its stored topology information, and a chosen shortest paths algorithm to determine the appropriate strand of MPLS labels corresponding to the output ports at each switch along the route. The controller then installs these rules on the edge switch as an instruction that will apply the actions sequentially. A final action will forward the resulting packet with all of its MPLS labels out the appropriate port. This process effectively encapsulates the packet such that subsequent switches will pop MPLS headers and forward their resultant packets based on the determined route.

Under this Nix-MPLS approach, the edge switches of the controlled network are the only points that must be provided new instructions while the remaining switches can be immediately prepared to match and route packets in a simpler fashion. This design differs significantly from typical SDN/OpenFlow flow rule installation. At best, most controller applications will simply preemptively push similar flow rules to each switch along a determined route when a new traffic flow is discovered. At worst, in applications that exhibit learning switch behavior, each switch along a route is tasked to request a flow rule for

any newly discovered traffic. For this work, the NIX-MPLS mechanism can be compared against the latter two rule installation processes across multiple controller libraries and shortest path algorithms, such as breadth first search (BFS), uniform cost search (UCS), and the Floyd-Warshall algorithm. In this way, a thorough examination of the performance of the NIX-MPLS approach can be achieved. Furthermore, the performance on specific controller libraries, such as Ryu and Floodlight, can be examined to demonstrate its effectiveness in different controller environments.

5.2 GPU-Based Routing with Parallel Floyd-Warshall

Since the controlled topology information in an SDN can be collectively determined and managed, route calculation can be accomplished using any shortest path algorithm. Providing scalable SDN topologies though can lead to routing algorithms that are not as scalable. Instead, it can be more appropriate to handle route computation in a parallel manner using the massively parallel processing capability of graphics processing units (GPUs). Using the parallelized version of the Floyd-Warshall algorithm introduced in [31], route generation can be accomplished in less time and with lower CPU utilization than with sequential graph algorithms.

The traditional Floyd-Warshall algorithm, originally described in [63], is an all-pairs shortest path algorithm that can be used to determine the length between any set of vertices in a weighted graph. The weights may be either positive or negative, although this advantage is less necessary in communication networks. The original algorithm alone cannot be used to discover the actual paths that have been calculated, but minor adjustments can remedy this issue. As shown in Algorithm 1, arrays are constructed such that their row indices correspond to source nodes and similarly the column indices represent the destination nodes of the weighted graph. All distances in the $N \times N$ array *dist* are initialized to ∞ unless a weight is already specified between a certain source and destination. For path determination through a next-hop examination, an additional array *next* is created to

ultimately designate through which node a path should travel to reach a destination. The algorithm then iterates through the array *dist* and attempts to find alternate paths between particular sets of nodes *i* and *j* whose distances are weighted less than the current distance between *i* and *j* (designated in the algorithm by separate links that connect through node *k*). If a shorter distance is discovered, this new value is stored for the length between *i* and *j*, and the path is updated by saving the next hop from *i* to *k* as the new hop from *i* to *j*. Through this modified version of the Floyd-Warshall algorithm, the shortest paths and the next hops to begin those paths can be determined by iterating through all combinations of *i*, *j*, and *k*.

Listing 5.1 demonstrates one potential kernel function that can be compiled to parallelize the Floyd-Warshall algorithm. The results calculated for particular combinations of *i* and *j* are independent from one another, and as such, their calculations may similarly be established independently from one another. However, the third iteration of the $1..N$ loop must remain as each new distance is calculated. Additionally, these calculations are permitted to occur in-place since the candidate distances and hops are initially stored and synchronized for each thread (through the `__syncthreads` CUDA function). If the threads are not synchronized at line 14 in Listing 5.1, slower threads may read incorrect data already modified by quicker threads. As discussed in Section 2.6, the potential for thread divergence exists as two conditional statements appear in the kernel. However, they pose little performance risk since no alternative paths accompany them.

For this work, the described implementation of the parallel Floyd-Warshall algorithm has been included in a controller application with path reconstruction and flow rule deployment occurring through instructions to each switch in a specific path. The application has been written using the Ryu controller library and the PyCUDA[64] CUDA API for Python. To better examine the performance achieved through the parallel implementation, benchmarks have been designed to allow the Floyd-Warshall algorithm to occur for each new `OFPT_PACKET_IN` message received by the controller. This performance is compared

against three alternate controller applications deploying the sequential Floyd-Warshall algorithm, BFS, and UCS.

A number of limitations in available hardware resources further constrains the testing and examination of the proposed SDN-based Floyd-Warshall deployment. Large-scale network testbeds, such as GENI or PlanetLab, do not readily provide virtual resources with explicit GPU computing capabilities. Furthermore, recent updates to the Linux kernel have resulted in IPv6 neighbor discovery behaviors that interfere significantly with recent versions of Mininet. This issue prevents a more thorough local examination utilizing this virtual testbed. Instead, the mechanisms introduced to DCE to provide Python[53] and CUDA[54] support supply an alternative simulated testbed for examining the network under the control of the aforementioned applications. For a more appropriate performance comparison, the real wallclock execution times (instead of the simulated times) in response to OFPT_PACKET_IN message events will be gathered for each controller application.

Algorithm 1 Floyd-Warshall with Next Hop

```

1: for  $i = 1..N$  do
2:   for  $j = 1..N$  do
3:     if  $W[i][j] \neq null$  then
4:        $dist[i][j] = W[i][j]$ 
5:        $next[i][j] = j$ 
6:     else
7:        $dist[i][j] = \infty$ 
8:        $next[i][j] = null$ 
9:     end if
10:  end for
11: end for
12: for  $i = 1..N$  do
13:   for  $j = 1..N$  do
14:     for  $k = 1..N$  do
15:       if  $dist[i][k] + dist[k][j] < dist[i][j]$  then
16:          $dist[i][j] = dist[i][k] + dist[k][j]$ 
17:          $next[i][j] = k$ 
18:       end if
19:     end for
20:   end for
21: end for

```

Listing 5.1: Kernel function for parallel Floyd-Warshall.

```
1 __global__ void fw(float *dist, int *next, int k, int N)
2 {
3     int i = blockDim.x * blockIdx.x + threadIdx.x;
4     int j = blockDim.y * blockIdx.y + threadIdx.y;
5
6     float check;
7     int next_;
8     if (i < N && j < N)
9     {
10         check = dist[j * N + k] + dist[k * N + i];
11         next_ = next[j * N + k];
12     }
13
14     __syncthreads();
15     if (i == 0 || j == 0 || i > N || j > N) return;
16
17     if (check < dist[j * N + i])
18     {
19         dist[j * N + i] = check;
20         next[j * N + i] = next_;
21     }
22 }
```

5.3 Introducing Validated Performance Modeling of SDN Simulation

As discussed in Section 2.3, most simulators and emulators that support SDN do not have validated controller models, and of the few that do, validation is primarily confined to the behavior of a single switch or a similarly low scale topology. This work aims to provide representative performance profiles that can introduce appropriate latencies and other behaviors into the SDN simulation framework discussed in Chapter 3. Using simple Ryu applications such as layer 2 forwarding as well as the proposed applications from Section 5.1, scalable network topologies such as linear and campus networks can be tested using both the hardware testbed GENI and network simulations. Controller processing time can be gathered and evaluated with the goal of working toward statistically similar results in both environments.

Baseline DCE processes the applications that it manages without affecting a change in the simulation time. For most simple applications such as *ping*, this lack of accuracy is ultimately negligible and can be ignored in the broader sense of the overall network simulation. However, for an application such as Ryu, which is being processed in Python and operating throughout the duration of the simulation, allowing the program to complete in zero-time can reduce its realism. The variety of processor architectures in production and the intricacies of how each one handles the execution of computer applications though makes it difficult and unnecessarily tedious to derive a closed-form solution to be used for influencing the timing results of a simulation model. Instead, application realism can be enhanced by experimentally determining an appropriate simulation model. A visually and statistically similar representation can be produced based on timing results gathered from real-world testbed experimentation. The frequency with which experimental timing results are observed produces discrete probability distributions that can be used to fit a random variable model. This model can in turn generate time values at frequencies resembling those observed during testbed experimentation.

Introducing the components that will elicit changes in the simulation time in DCE requires an understanding of how DCE currently manages its applications. As described in Section 2.2, DCE is composed of a core, kernel, and POSIX layer. The core layer abstracts and maintains all of the components of a particular DCE application in a way that it can be handled by the ns-3 event scheduler. The POSIX layer interacts with the core layer in a way that allows both the natively managed and DCE-handled calls to *glibc* to become events that can be scheduled. These events are ultimately packaged in objects called *tasks* which are separately maintained in DCE by a *task manager*. Tasks can be instantiated and then placed in any of the following states based on which process in a DCE application is currently running: ACTIVE, RUNNING, BLOCKED, or DEAD. When a task is first created, it is set to BLOCKED, forcing the task manager to wake it up in order to actually place it in the ns-3 event scheduler. The core layer is responsible for prompting the task

manager to wake up certain tasks as application processes are swapped between sleeping and active states. In its current design, DCE accomplishes this scheduling in an immediate manner without allowing simulation time to advance. When a task event is executed, it will be set to the RUNNING state. The task can then invoke its event which is a *glibc* function call. This operation will iterate as long as task-related events are available and the associated process is running.

The network simulator ns-3 provides a number of options for generating random variables based on probability distributions. By introducing a randomized delay based on experimentally determined distribution parameters, the tasks can appear to incur a realistic time cost when they are invoked. However, certain operations such as basic assignment or simple arithmetic or logical procedures do not involve *glibc* in the way that DCE could manage them. These operations will only complete in zero simulation time and cannot be explicitly influenced by a simulated delay model without significant modification to the current design of DCE. With this limitation in mind, the influence of those tasks that can be managed by DCE may need to be enhanced in such a way that they can additionally capture some of the timing effects that cannot be readily simulated.

CHAPTER 6

EXPERIMENTAL SETUP

Experiments for examining both the efficacy of the proposed routing mechanisms in SDN as well as the adequacy of the validated model for the SDN simulation framework require a multi-faceted approach. In order to determine the appropriate characteristics and parameters of the simulation model, experiments must first be performed within a real-world network testbed to gather representative data. The network testbed utilized for this work is GENI[65]. These experiments provide the two-fold advantage of providing data relevant to the controller processing times while also producing results that can be used to gauge the performance of the proposed routing mechanisms.

6.1 Network Topologies

Two network topologies have been constructed in both GENI and the network simulator ns-3 to examine and compare the performance of the controller applications and their routing policies described in Chapter 5. A linear network of switches is studied to provide a simple yet extensible example demonstrating baseline functionality of each of the proposed routing schemes. The Network Management System (NMS) challenge topology, proposed by Nicol in [66], is considered as well to gather results from what is not only a broadly utilized benchmark topology [67, 68] but also a representative model of an adequately complex and relevant network topology. As appropriate, necessary differences in topology and traffic design between GENI and simulation are provided. As an assumption, links between nodes in either GENI or ns-3 are considered 1Gbps, and for ns-3 specifically, delays are 1ms as it is difficult to deterministically verify specific delays from the GENI experiments.

6.1.1 Linear Network

The linear network of switches, depicted in Figures 4.3 and 6.1, demonstrates baseline functionality of the proposed controller applications. As its name implies, the linear network is a set of 8 switches each connected to one another to form a single line. Each switch is then connected to its own set of end hosts. Each switch also connects to a single controller node. In GENI, in order to reduce the number of total VMs requested, each switch connects to only one host each but is provided 8 distinct links between itself and its host node. In this way, multiple addresses (both Ethernet and IP) are made available for traffic generation. This design in turn permits additional flow rules to be installed on the various switches, building an examination of the routing schemes in the presence of more representative flow table lookups. Request limitations for simulation are internally dependent on the resource constraints of the utilized computer system rather than externally dependent on resource sharing among multiple users. As such, simulations of the linear network performed in ns-3 are not constrained in the same way as in GENI so each switch is connected to its own distinct set of 8 hosts.

6.1.2 Campus Network

The campus network, shown in Figures 6.2 and 6.3 is based on the NMS challenge topology proposed by Nicol in [66]. This topology is intended to resemble a generic campus network and consists of four subnetworks as well as two connective middleboxes. In traditional network examples, the rectangular middleboxes depicted are routers that provide connectivity. However, for this work, each middlebox is an OpenFlow-enabled switch, and as shown in Figure 6.3, each switch connects to a single controller node. The yellow circles from Figure 6.2 are normally implemented as local area networks (LANs) with a variable number of hosts. Similarly to the design choice made for the linear network, in GENI, each LAN is simply a single host VM with 8 links to its switch; in ns-3, 8 distinct hosts link to a particular switch. The campus topology itself is composed of the connective subnetwork



Resources on Georgia Tech Instageni are ready.

View Rspec

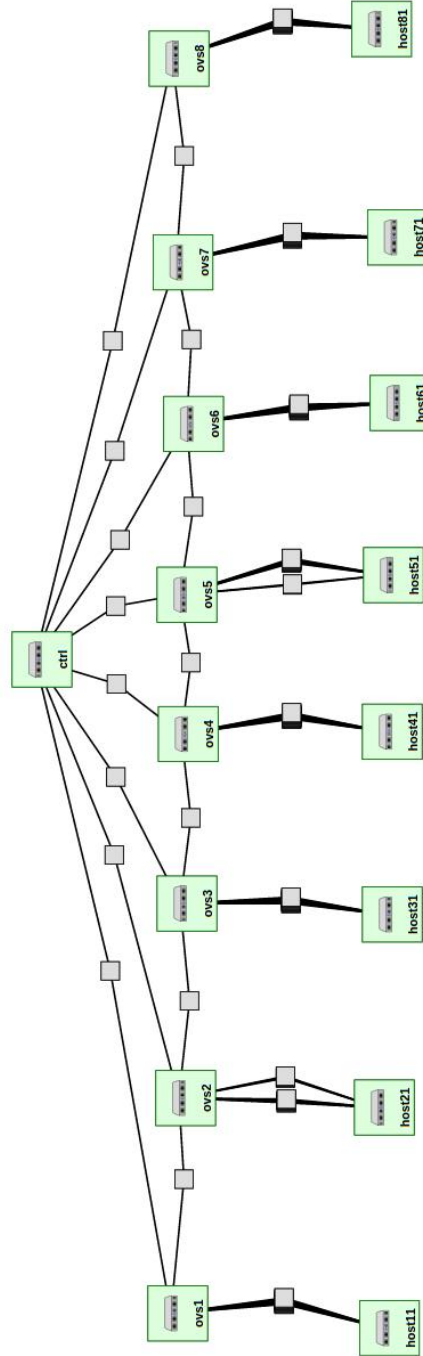


Figure 6.1: Linear network in GENI

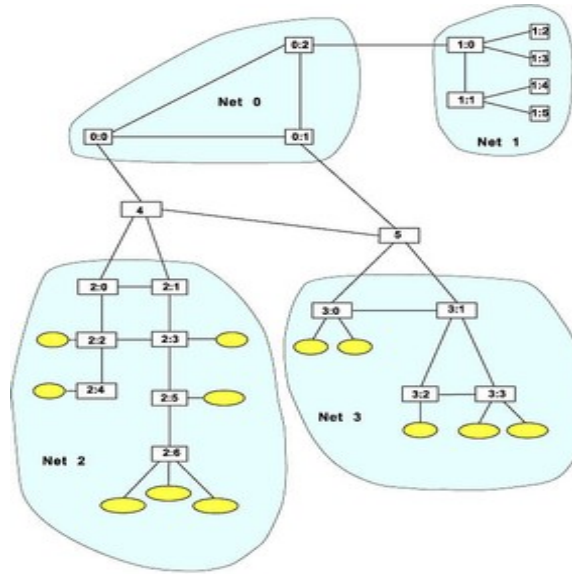


Figure 6.2: Nicol NMS challenge topology

Subnet 0, Subnet 1 holding VMs acting as servers, and Subnets 2 and 3 which respectively have 7 and 4 switches with some of those switches connected to client LANs. Switches 4 and 5 connect Subnets 2 and 3 respectively to Subnet 0 as well as to one another.

Ring of Campus Networks

The previously discussed networks examine only 8 and 18 switches. This constraint is enforced for the examination of the Nix-MPLS performance and the controller processing modeling effort due to the shared nature of the GENI resources. Larger networks can be achieved in simulation and are advantageous for thoroughly examining the performance of CUDA-based routing. One method for achieving large-scale topologies is through simple replication of the campus network topology. These replicated campus networks can then be connected to one another at Switch 0 in Subnet 0, forming a ring of campus networks. Within the context of the Floyd-Warshall algorithm, this extension of the single campus network increases the $N \times N$ arrays by a factor of 4 for each new campus, resulting in 8 times as many computations that must occur.



Resources for Slice geni-desktop-slice

Resources on Kettering Instageni are ready.

View Rspec

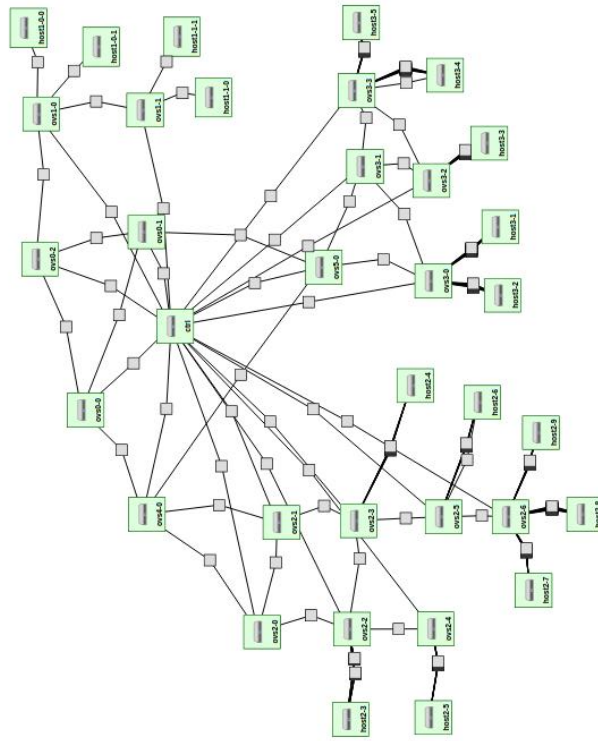


Figure 6.3: Campus network in GENI

6.2 Traffic Generation and Observed Variables

The *ping* command is used to study the RTT of some of the longest trips a packet can make in the investigated networks. For the linear network, this trip is the connection between the host on switch 1 to the host on switch 8. For the campus network, three separate *ping* series are examined. One instance simply checks the RTT between a host on the first switch in subnet 1 to a host on the second switch in subnet 1. Another set examines the RTT between a host on the furthest switch in subnet 2 to the host on the second switch in subnet 1. The final set similarly studies the RTT between a host on the furthest switch in subnet 3 to the host on the second switch in subnet 1. For the ring of campus networks, a similar scheme is employed with just three *ping* series, but the senders reside on the first campus, and the recipient sits in the campus halfway across the ring.

Two separate instances of the *ping* command are studied for each series in both the linear and campus networks. Studying two *ping* requests allows both the controller and network performance to be examined. For the first *ping*, switches have not yet learned the routes and must be provided the appropriate flow rules by the controller. From this *ping*, the performance of each controller application is studied as it relates to both how effectively the controller disseminates flow rules and how quickly each switch can in turn forward the initial packet based on these rules. A second *ping* is then sent after the first one has completed in order to examine the performance of the network after having learned the appropriate routing scheme. Without having to consult the controller, the switches are now tested on how well they perform flow table lookups based on the routing schemes and subsequently forward the packet. For the simulated network, ICMP packets are sent similarly through the built-in `V4Ping` application.

Generating traffic within the network provides resource contention, subjecting the controller applications and the controlled switches to increasing loads under which they must make routing decisions for the network. In the case of the linear network, these routing

decisions are not particularly interesting; however, their performance is still necessary for study prior to examination under more complex topologies. For the linear network studied in GENI, traffic is generated using the *iperf* command. UDP traffic is continuously sent at 100Kbps from a particular IPv4 address on one host to a particular IPv4 address on another. UDP datagram size is set to 1400 bytes to provide sufficient space for the addition of multiple MPLS labels. (Alternatively, the maximum transmission unit (MTU) for each NIC in the network could be set larger, but this configuration variability is left to future study.) For the simulated network, traffic is generated using the `OnOffApplication` provided by ns-3. This application provides a similar behavior to *iperf* in terms of setting traffic protocol, packet size and bandwidth. The ICMP packets transmitted during the *ping* commands are set to 1400 bytes as well to better mimic the UDP traffic behavior.

Under various network loads, the effective throughput of the system can be calculated to gauge how well each controller application and its respective routing scheme handles increasing levels of network traffic. Specifically for the experimental setup of the linear network, traffic flows are distinguished in the following ways:

- None: Only the *ping* traffic is observed.
- Small (4 flows total):
 - Switch 2 to 7: UDP traffic sent from 4 addresses (hosts in simulation) on one switch to the hosts on another.
- Medium (8 flows total):
 - Small traffic
 - Switch 3 to 6: UDP traffic sent from 4 addresses on one switch to the hosts on another.
- Large (12 flows total):

- Small traffic
- Medium traffic
- Switch 4 to 5: UDP traffic sent from 4 addresses on one switch to the hosts on another.

The experimental setup of traffic generation in the campus network is designed in the following ways:

- None: Only the *ping* traffic is observed.
- Small (16 flows total):
 - Switch 3, Subnet 2 to Switch 1, Subnet 1: UDP traffic sent from 4 addresses (hosts in simulation) on one switch to host 1 on another.
 - Switch 4, Subnet 2 to Switch 1, Subnet 1: UDP traffic sent from 4 addresses on one switch to host 2 on another.
 - Switch 1, Subnet 3 to Switch 2, Subnet 1: UDP traffic sent from 4 addresses on host 1 on one switch to host 1 on another.
 - Switch 1, Subnet 3 to Switch 2, Subnet 1: UDP traffic sent from 4 addresses on host 2 on one switch to host 2 on another.
- Medium (32 flows total):
 - Small traffic
 - Switch 5, Subnet 2 to Switch 1, Subnet 1: UDP traffic sent from 4 addresses on one switch to host 1 on another.
 - Switch 6, Subnet 2 to Switch 1, Subnet 1: UDP traffic sent from 4 addresses on one switch to host 2 on another.
 - Switch 3, Subnet 3 to Switch 2, Subnet 1: UDP traffic sent from 4 addresses on one switch to host 1 on another.

- Switch 4, Subnet 3 to Switch 2, Subnet 1: UDP traffic sent from 4 addresses on host 1 on one switch to host 2 on another.
- Large (48 flows total):
 - Small traffic
 - Medium traffic
 - Switch 7, Subnet 2 to Switch 1, Subnet 1: UDP traffic sent from 4 addresses on host 1 on one switch to host 1 on another.
 - Switch 7, Subnet 2 to Switch 1, Subnet 1: UDP traffic sent from 4 addresses on host 2 on one switch to host 2 on another.
 - Switch 4, Subnet 3 to Switch 2, Subnet 1: UDP traffic sent from 4 addresses on host 2 on one switch to host 1 on another.
 - Switch 7, Subnet 3 to Switch 2, Subnet 1: UDP traffic sent from 4 addresses on host 3 on one switch to host 2 on another.

The ring of campus networks is only studied with the *ping* traffic and no additional traffic flow since it has only been examined within simulation.

Because the focus of this work is to assess the performance of the network in relation to the controller applications under investigation, packet flow through the network is limited to *ping* and *iperf* traffic and the LLDP packets that the controller uses for topology discovery. The nature of the experiments described leaves little extraneous packet flow. However, one significant remaining traffic type is ARP. ARP requests and replies are transmitted from end nodes initially when they have yet to associate destination IP addresses with Ethernet addresses. This traffic impacts the controller as well since it has most likely not learned which end hosts are associated with which switch ports in its network. Since it does not possess this information *a priori*, the controller cannot instruct the switches to forward packets out of the network directly to the appropriate recipient. Instead, ARP

requests must be flooded out of the network under the assumption that an ARP reply will get returned subsequently from the associated switch port. If an ARP reply is returned, the controller now knows of both end hosts and from which switch ports their packets arrive, allowing a complete path within the network to be determined. Rather than allowing the network traffic to be overrun by ARP packets while performance is being monitored, the network is “primed” prior to data collection through the use of the *arping* program. This application operates similarly to the *ping* command, but instead of ICMP packets, it only transmits ARP requests and replies. In this way, the controller can manage ARP flooding without receiving any other packets that may allow it to learn actual routes prematurely (in terms of the experimentation). It is also allowed to handle that flooding without interfering with the processing of other traffic. A simple `ArPing` application has been written as well for ns-3 to provide similar functionality within simulations.

CHAPTER 7

RESULTS AND DISCUSSION

7.1 SDN-Based Nix Vector Routing

Experiments performed on the linear and single campus networks have been examined in GENI to investigate the real-time effects of the Nix-MPLS routing behavior against traditional flow installation schemes. GENI aggregates at the Georgia Institute of Technology (Georgia Tech) and Kettering University provide the compute resources for the linear and campus networks, respectively. For both aggregates, the controllers, switches, and end hosts comprising the two topologies are implemented with InstaGENI as Xen VMs running Ubuntu 14.04. Each VM is provided access to a single CPU core of a hex-core 2.67GHz Intel Xeon X5650 processor. The GENI aggregate at Georgia Tech provided 499MB of memory for each VM in the linear network. The VMs constituting the campus network at Kettering were each provisioned with 368MB.

For these experiments, the Nix-MPLS scheme has been introduced in both the Ryu and Floodlight controller libraries in order to demonstrate and compare its applicability in both environments. Specifically, versions of the Ryu and Floodlight controllers used in these experiments are 4.10 and 1.2, respectively. The 2.6.2 version of the OpenvSwitch library is installed on the VMs acting as switches in each network. A minor patch is included to extend the number of MPLS labels that can be pushed by OpenvSwitch from 3 to 14. This work focuses on the OpenFlow 1.3 specification. Experiments have been executed 20 times across each routing and traffic generation scheme to determine their resulting average RTT and effective throughput. An additional variable is studied as well which demonstrates the performance of the traditional flow installation schemes in both user space and kernel space (denoted by (K) in the resulting figures). Kernel space OpenvSwitch could not be used in

conjunction with NIX-MPLS since it cannot process the pushing and popping of MPLS headers.

Flow installation for a particular traffic flow differs slightly between the implementations in the Ryu and Floodlight controllers. In Ryu, since the applications have been written from scratch, forward and reverse paths are calculated for both simple flow rule installation and NIX-MPLS. This design further limits the amount of communication through `OFPT_PACKET_IN` messages between the controller and switches, improving transmission times and bandwidth. In contrast, since NIX-MPLS flow rule installation is introduced in Floodlight as a patched modification to the baseline Forwarding application, only forward paths are installed for each new traffic flow to mimic the baseline implementation for a balanced comparison.

7.1.1 Linear Network

For the linear network experiments, multiple controller applications have been examined. In Floodlight, the Layer 2 Learning and Forwarding baseline applications are studied against the patched Forwarding application that uses NIX-MPLS flow rule installation. In Ryu, its baseline *simple_switch_13* application which accomplishes layer 2 learning is studied as well as applications that demonstrate either BFS, UCS, or Floyd-Warshall for path determination and either simple or NIX-MPLS behavior for flow rule installation.

Floodlight

The results for the linear network experiments running Floodlight are shown in Figures 7.1, 7.2, and 7.3. Standard error bars are displayed but are generally noted as negligible aside from some of the results for simple layer 2 learning. For the linear network running the Floodlight controller, major differences in performance are primarily noted in Figure 7.1 for the initial *ping* RTT results. The general lack of variation seen in Figures 7.2 and 7.3 can be seen as more of a testament to the robust nature of Floodlight rather than a reason

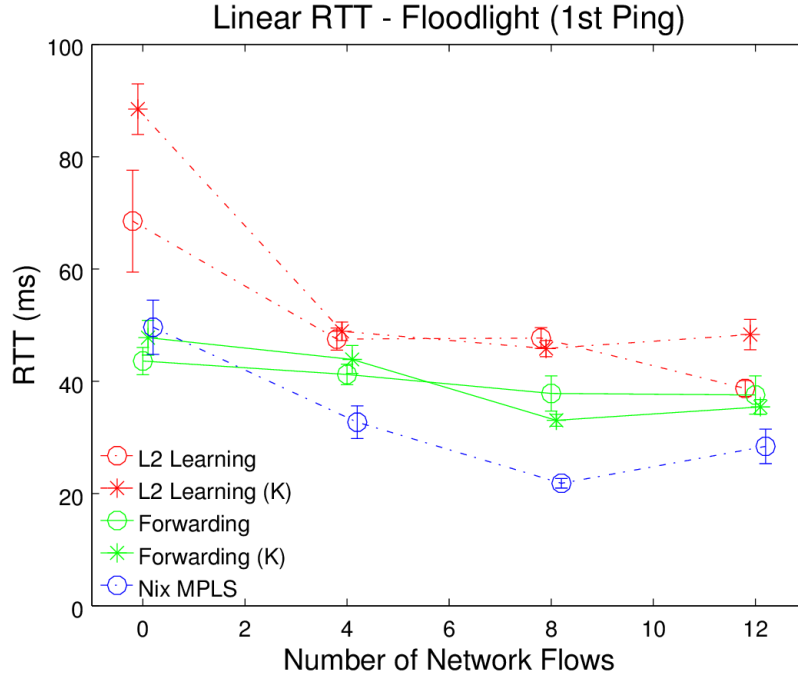


Figure 7.1: Floodlight RTT (1st Ping) - Linear Network

against supporting the Nix-MPLS behavior. By maintaining adequate load balancing over basic BFS path determination, the Floodlight controller is able to provide sufficient packet routing regardless of the flow installation mechanism.

Initial flow learning, as demonstrated by the results in Figure 7.1, provides the most distinct variation seen across the variables tested. Layer 2 learning provides the slowest flow installation, resulting in slower reported RTTs whether it is executed in kernel or user space. As discussed in Section 2.1.3, switches running the kernel space configuration for OpenvSwitch must still communicate through the user space upon initial flow installation. This process ultimately causes the performance of the kernel space variant to approximately mirror the results when a switch is operating in user space. In some cases, as noted in the figure, this process can potentially introduce an overhead when a switch must decide to forward a packet from kernel space to user space rather than simply transmit the packet to user space by default. Additionally, the decrease in RTT seen from layer 2 learning to the default Forwarding application to the Nix-MPLS installation behavior is the most evident

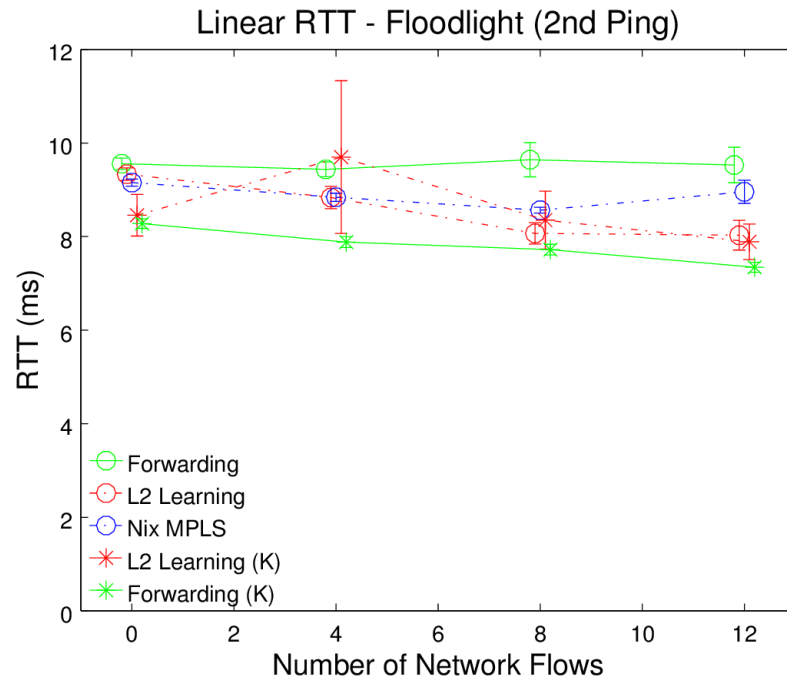


Figure 7.2: Floodlight RTT (2nd Ping) - Linear Network

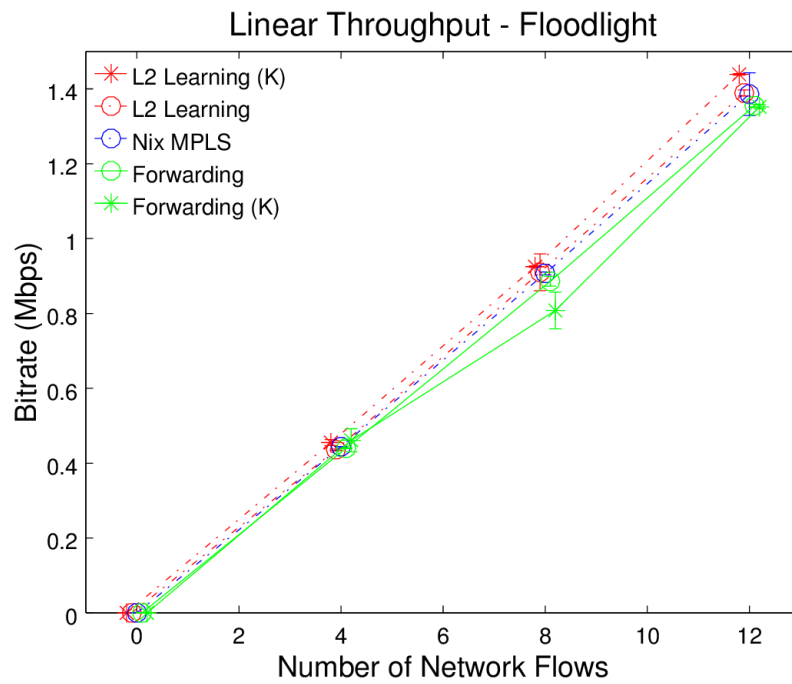


Figure 7.3: Floodlight Throughput - Linear Network

demonstration of the reduction in flow rule installation. For layer 2 learning, each switch will receive the initial packets from a particular traffic flow without having any matching flow rules installed. As such, each new packet will result in an `OFPT_PACKET_IN` message that will be received by the controller for each switch. The controller must then transmit the flow rules to each switch. This behavior becomes less burdensome to the controller as additional traffic flows allow it to cache information for later flows. For the Forwarding application, the controller will preemptively install flow rules for each switch in a determined path, limiting the number of `OFPT_PACKET_IN` messages but still requiring the same number of flow installations. For Nix-MPLS, flow rules within the switch network are preemptively installed when the switch first connects to the network. In this way, only one edge switch interacts with the controller when a new traffic flow is discovered, limiting the resultant network traffic between the controller and its switches.

Ryu

The results for the linear network experiments running Ryu are shown in Figures 7.4, 7.5, and 7.6. Compared to the results under the Floodlight controller, the impact of employing different applications is more discernible in the Ryu controller. The combination of differences in the designs of these two controller libraries influences the resulting discrepancies. Written in different programming languages, these two controllers are designed with different threading mechanisms and event handler libraries. Similarly, although both Python and Java implementations are ultimately referencing C libraries, the wrapping of these libraries for each particular language differs such that other components like sockets are handled somewhat differently. In turn, the resulting controller libraries differ architecturally, producing different behaviors.

Under the first *ping* command, considerable differences can be observed. Layer 2 learning generally exhibits the highest RTT when initially learning flow rules since it must communicate those rules to each individual switch in the path as each one discovers the

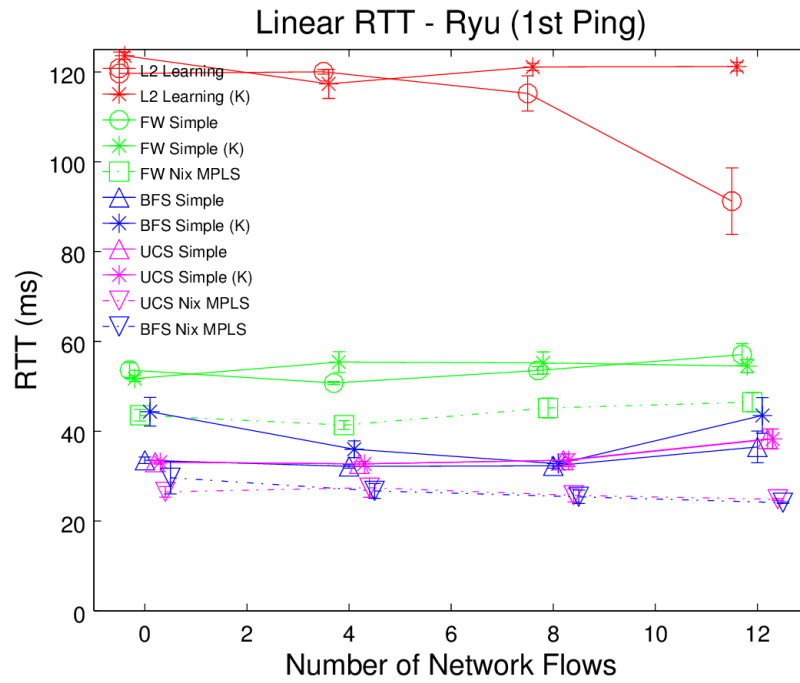


Figure 7.4: Ryu RTT (1st Ping) - Linear Network

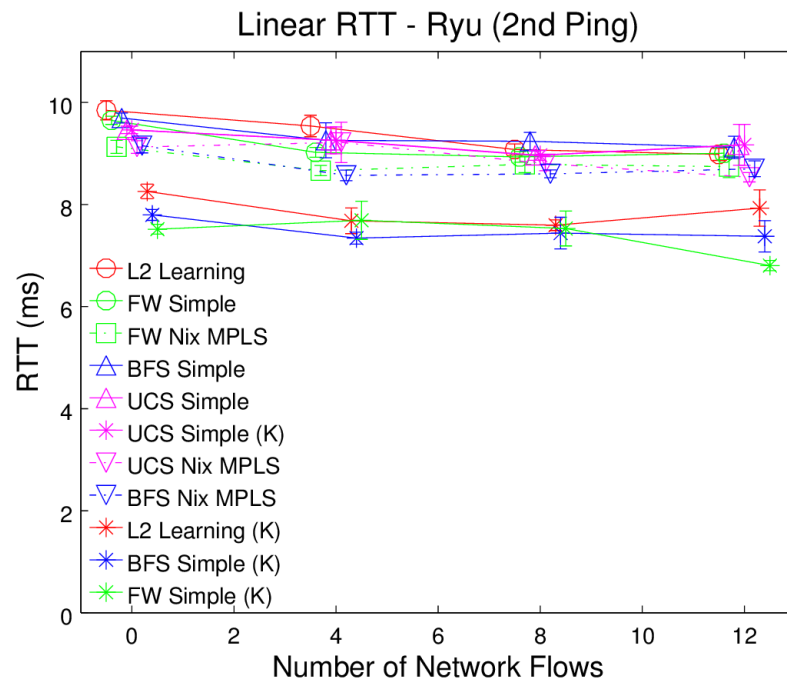


Figure 7.5: Ryu RTT (2nd Ping) - Linear Network

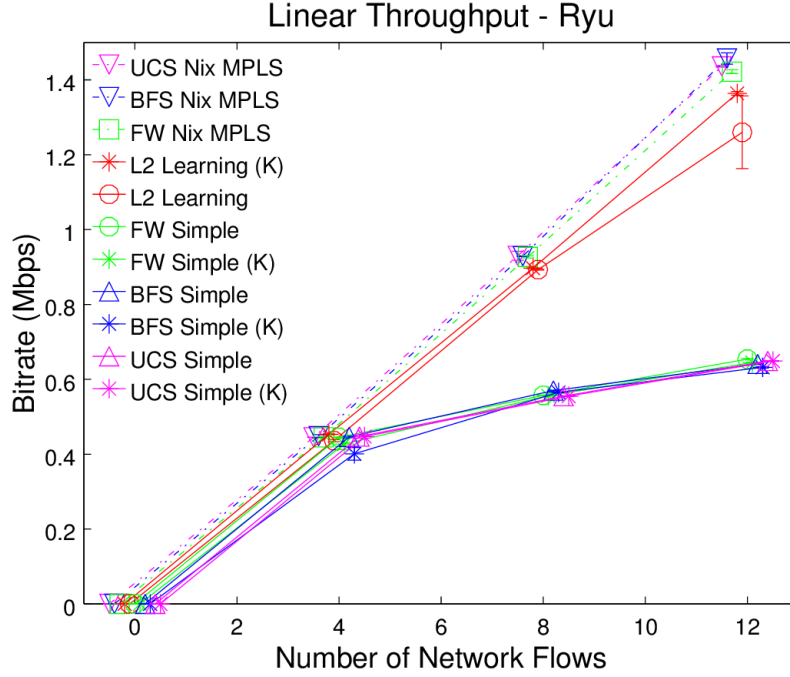


Figure 7.6: Ryu Throughput - Linear Network

new traffic flow. Floyd-Warshall exhibits improvements with the Nix-MPLS flow installation. Likewise, BFS and UCS show improvement when this alternative flow installation is deployed. Furthermore, BFS and UCS provide a lower RTT than Floyd-Warshall since the former two algorithms search for one particular path while Floyd-Warshall calculates all routes. Little difference is seen between the results of BFS and UCS since the linear network only provides one possible route for every source-destination combination. The reduction in RTT noted in response to the initial *ping* packet derives from the same reasoning addressed in the discussion of the Floodlight controller. Under the Nix-MPLS concept, only one rule at the edge switches needs to be installed for a new traffic flow, limiting the degree of coordination between the controller and switches. The second ICMP packet transmissions produce similar results to the RTTs observed under the same circumstances in the Floodlight controller. Since the switches have learned their flow rules and do not require interaction with the controller in the context of these particular flows, the influences of controller library choice become less distinguishable. Results from the experiments em-

ploying kernel space OpenvSwitch provide lower RTTs for the second *ping*, demonstrating its general efficacy over its user space variant.

A greater advantage of deploying the NIX-MPLS flow installation can be noted by the effective network throughput shown in Figure 7.6. The packet event handling in Python and consequently Ryu is less efficient than the handling in Java and Floodlight. As such, increasing traffic flows will produce more OFPT_PACKET_IN traffic at the controller. This subsequent increase in traffic leads to the overall reduced throughput displayed in Figure 7.6. The fewer OFPT_PACKET_IN messages required for NIX-MPLS decreases the burden on the controller, permitting the network to obtain a higher effective throughput.

7.1.2 Campus Network

For the campus network experiments, a subset of the controller applications considered for the linear network have been examined. The controller applications exhibiting layer 2 learning are not assessed since they do not provide mechanisms to prevent forwarding loops. In Floodlight, the unpatched and patched Forwarding applications are studied. In Ryu, applications demonstrate either BFS, UCS, or Floyd-Warshall for path determination and either simple or NIX-MPLS behavior for flow rule installation.

Floodlight

The results for the campus network experiments running Floodlight are shown in Figures 7.7, 7.8, 7.9, 7.10, 7.11, 7.12, and 7.13. In the case of the RTTs viewed for Subnet 1, any observed differences are minimal. This similarity though is reasonable since packets are only traversing a single link in the controlled network. The reduced number of decision points in the network leads to less processing that is required when forwarding decisions are made. In this same way, a minor improvement in the RTT for the first *ping* under Forwarding using kernel space OpenvSwitch reduces the impact of the decision for forwarding between kernel and user space when no other network traffic is occurring. This processing

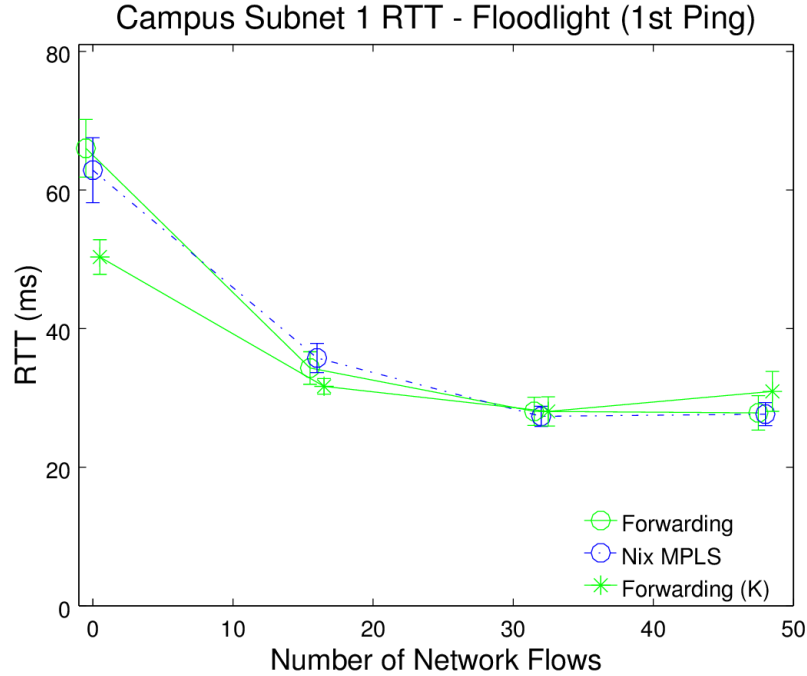


Figure 7.7: Floodlight RTT from Subnet 1 (1st Ping) - Campus Network

reduction is less evident after flows have been installed, as noted in Figure 7.10. As shown in Figures 7.8 and 7.9, the first ICMP packets transmitted from Subnets 2 and 3 result in somewhat inconclusive results. Due to the uncertain nature of executing the experiments on real hardware, variability between each experimental attempt, in addition to varied network traffic, can result in inconsistent trends when observing the average data results. This point can be seen in Subnet 3 especially by the response of the kernel space OpenvSwitch deployment to increasing network traffic under Forwarding control. It can also be noticed less prominently in Subnet 2 for the two user space OpenvSwitch experiments.

The second *ping* transmissions in both Subnets 2 and 3 exhibit little difference across the two flow installation behaviors when user space OpenvSwitch is deployed. This similarity continues to be a result of the additional mechanisms at work within the Floodlight controller. Kernel space OpenvSwitch deployment, as has previously been noted, continues to produce stable RTT improvement as it coordinates with the mechanisms of the Floodlight controller. Reduction in RTT with increasing network traffic across all observations can be

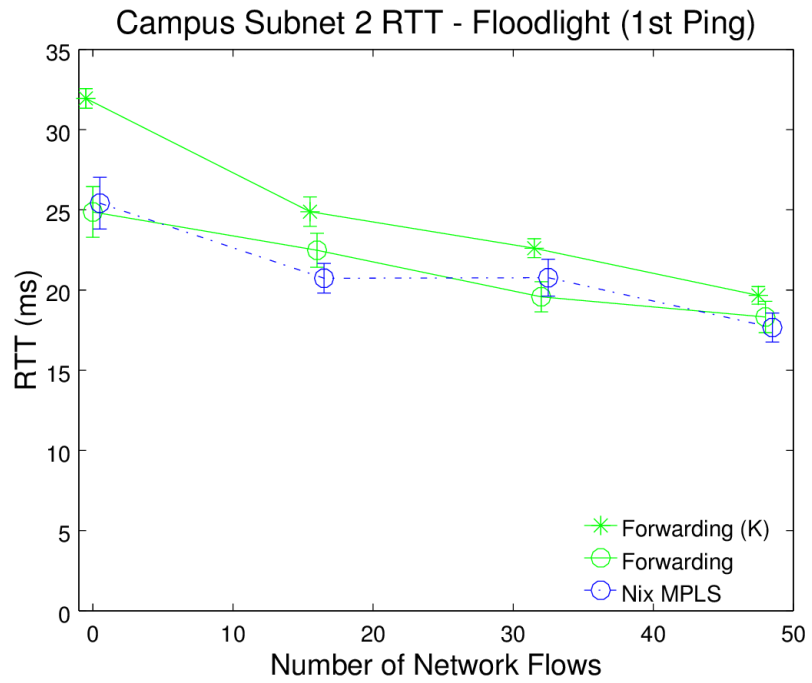


Figure 7.8: Floodlight RTT from Subnet 2 (1st Ping) - Campus Network

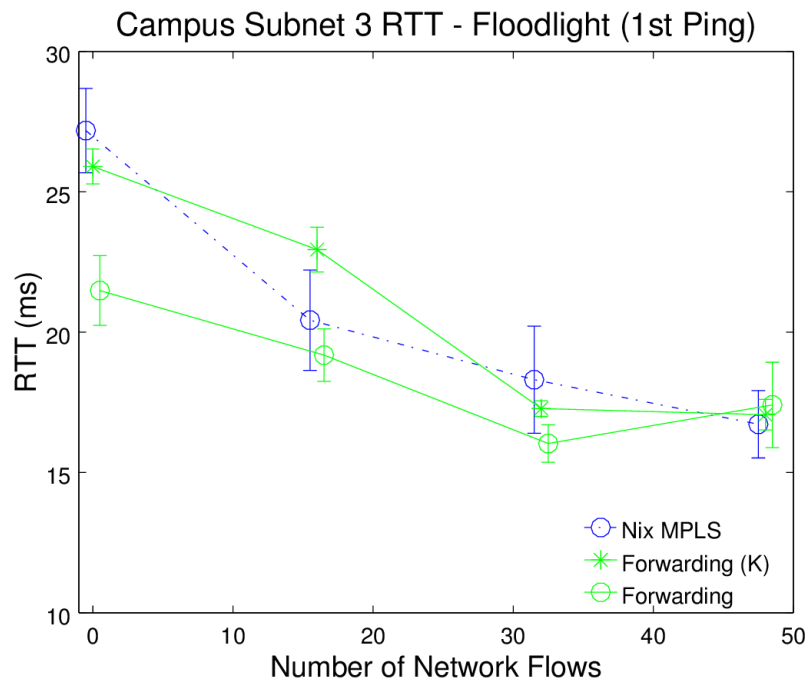


Figure 7.9: Floodlight RTT from Subnet 3 (1st Ping) - Campus Network

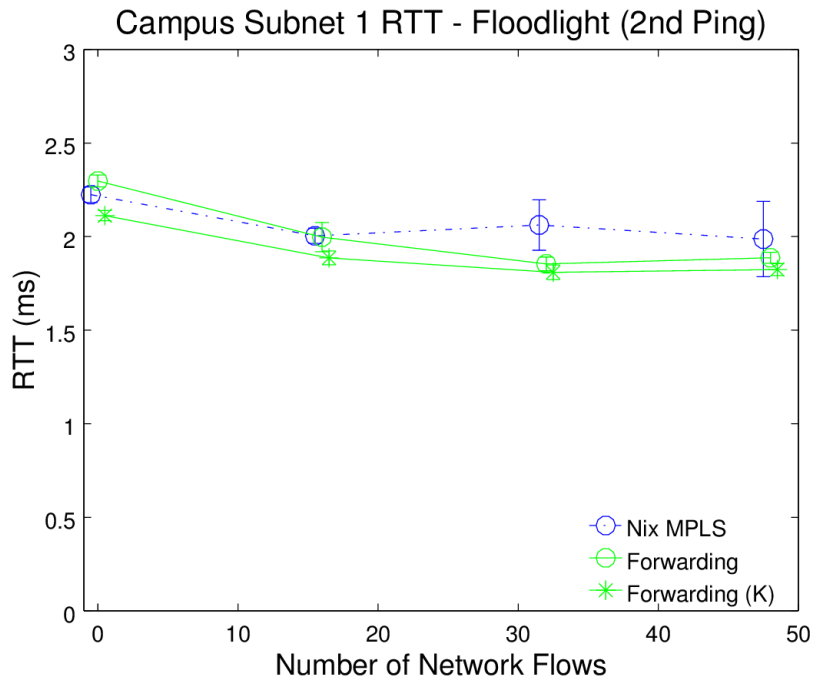


Figure 7.10: Floodlight RTT from Subnet 1 (2nd Ping)- Campus Network

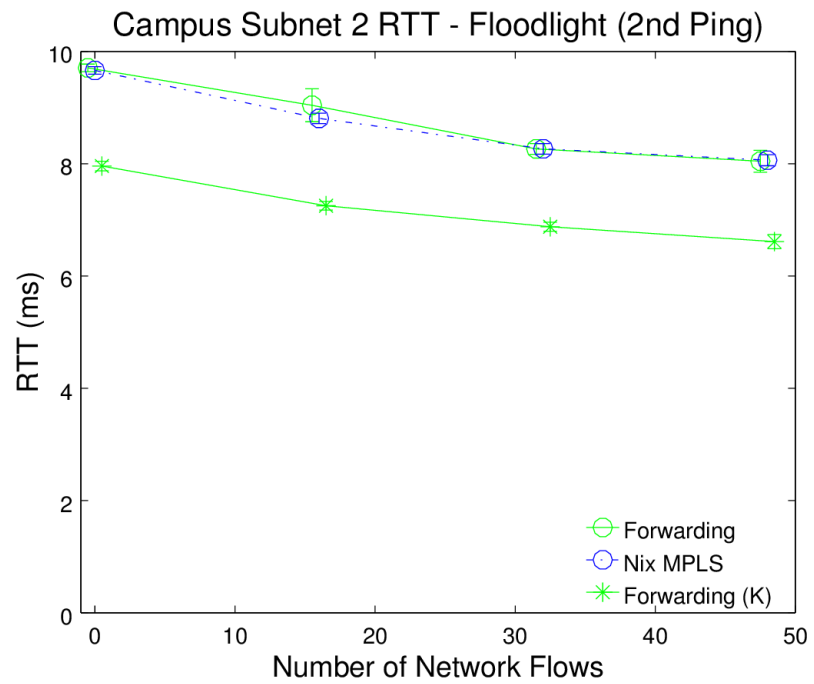


Figure 7.11: Floodlight RTT from Subnet 2 (2nd Ping)- Campus Network

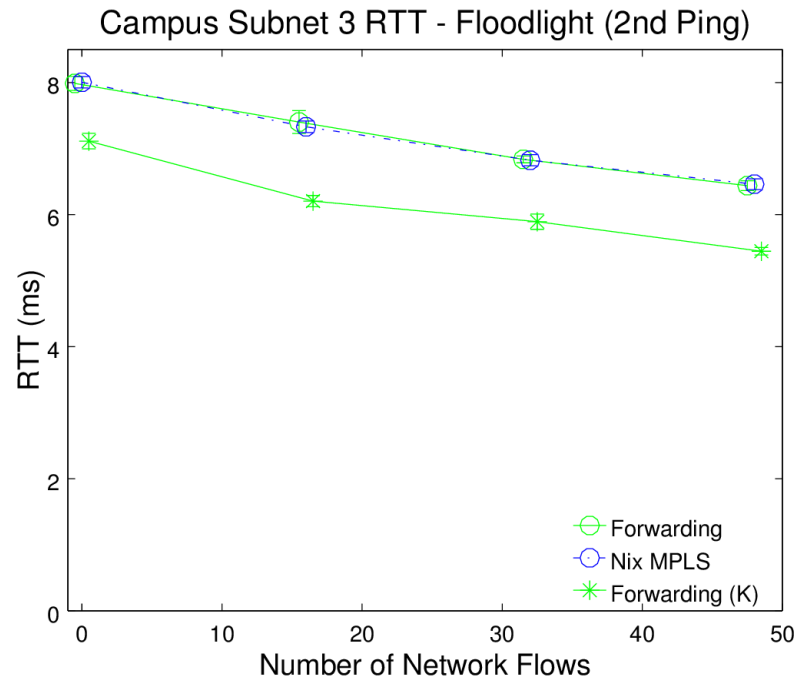


Figure 7.12: Floodlight RTT from Subnet 3 (2nd Ping)- Campus Network

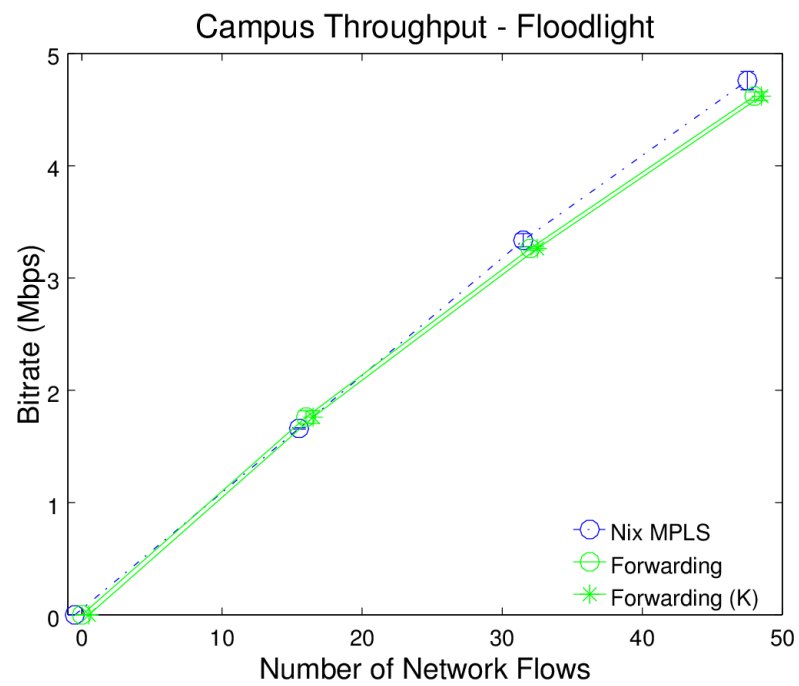


Figure 7.13: Floodlight Throughput - Campus Network

generally attributed to the combination of load balancing and flow information caching administered in Floodlight. As shown in Figure 7.13, improvement in effective throughput can be noticed for the Nix-MPLS behavior, especially with increasing network traffic. However, the increase is minimal, and further testing would be beneficial to determine if the trend might continue with the addition of more network traffic.

Ryu

The results for the campus network experiments running Ryu are shown in Figures 7.14, 7.15, 7.16, 7.17, 7.18, 7.19, and 7.20. General trends among the variables tested are relatively inconsistent, most likely due to the degree of uncertainty associated with executing these experiments in real-time. What can be seen for most levels of traffic though is the ability for BFS and UCS with Nix-MPLS to produce the lowest RTT for the first ICMP packet transmissions from any subnet. Additionally, for most traffic flow cases in Subnets 2 and 3, the RTT results for the Floyd-Warshall algorithm are improved with Nix-MPLS. BFS and UCS continue to demonstrate approximately similar behavior, although some cases also resulted in higher RTTs from UCS. The inability for UCS to produce improved results to BFS by calculating the lowest-cost paths can be attributed to a relative lack of path diversity and the fact that all observed paths must still traverse a single link between Subnets 0 and 1. Although more paths are available compared to the true lack of possibilities in the linear network, it can be assumed that the traffic patterns selected would task the various routes equally. In this case, the additional processing of UCS to determine lowest-cost paths would simply become computational overhead. Similarly to most other cases observed, the RTT values from the second *ping* commands are lowest when kernel space OpenvSwitch is used since flow rules have been installed and the switches can simply process packets completely in kernel space.

As shown in Figures 7.15, 7.16, 7.18, and 7.19, the highest level of traffic (48 network flows) produces inconsistent behavior in Subnets 2 and 3. As additional traffic is managed

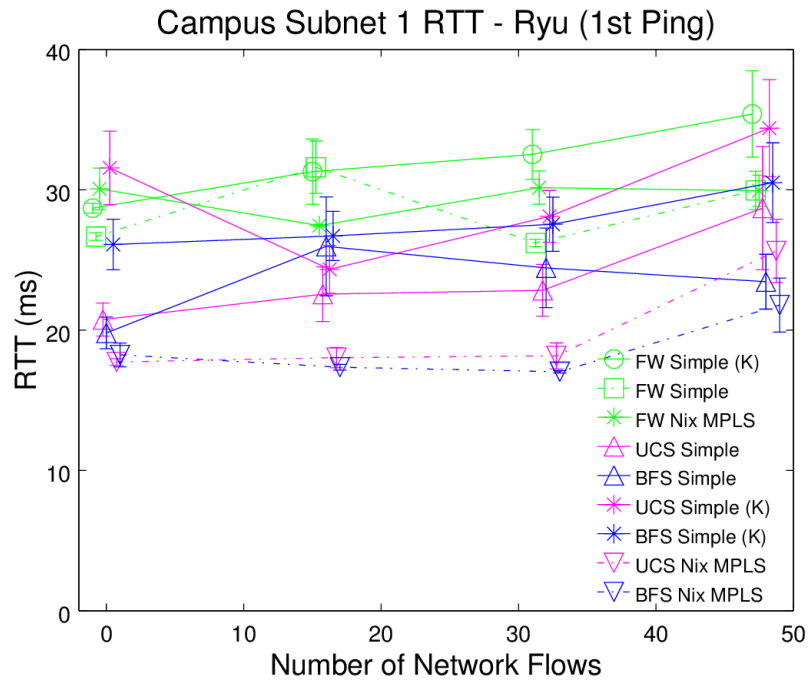


Figure 7.14: Ryu RTT from Subnet 1 (1st Ping) - Campus Network

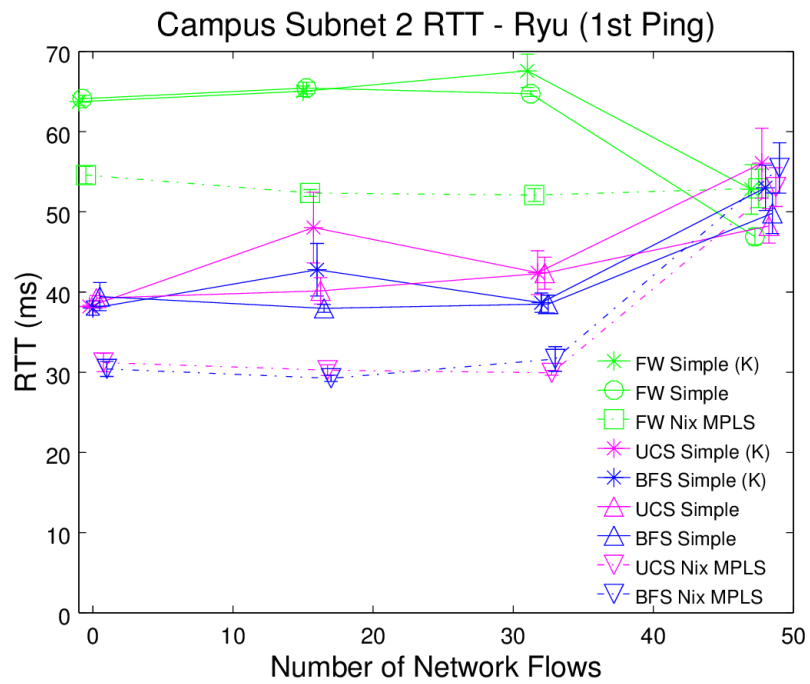


Figure 7.15: Ryu RTT from Subnet 2 (1st Ping) - Campus Network

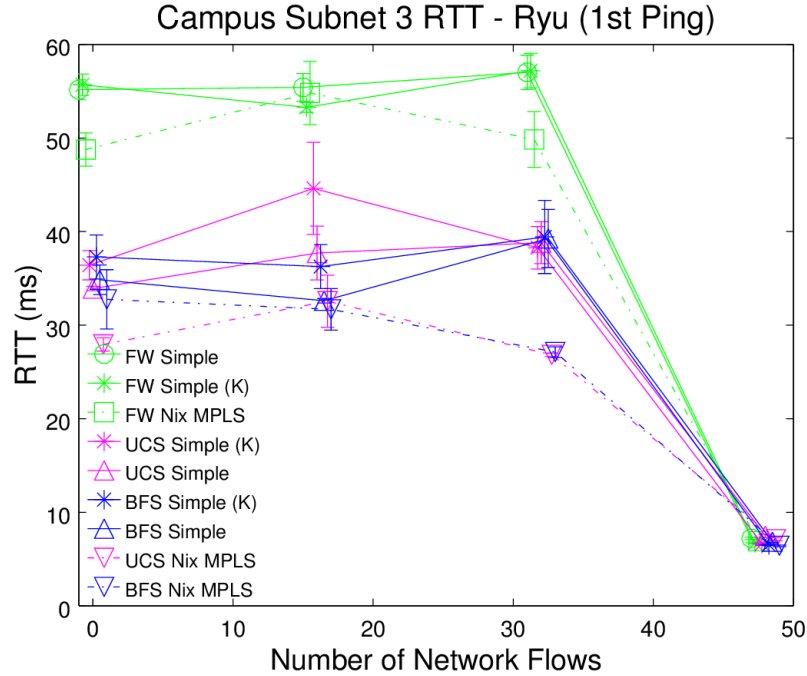


Figure 7.16: Ryu RTT from Subnet 3 (1st Ping) - Campus Network

and resources are contended within Subnet 0 to move packets between the client Subnets and Subnet 1, it can become more difficult for packets to traverse the network. This point is especially noticeable for packets transmitted from the farthest areas of Subnet 2 where a packet may potentially encounter over 10 decision points within the network, a possibility less likely for packets from Subnet 3. Each decision point in turn introduces a point in the path for a packet where it may get dropped under resource contention, requiring a retransmission from its source for ICMP packets. This drawback in Subnet 2 consequently promotes an environment in Subnet 3 from which packets are less likely to be dropped and require retransmission. This notion could explain the resultant drop observed in RTT for the first *ping* transmissions seen in Figure 7.16 and the abrupt increase for the second transmissions seen in Figure 7.18.

As shown in Figure 7.20, effective throughput is again found to be improved when Nix-MPLS flow installation is used. The same reason discussed in Section 7.1.1 applies for the campus network as well where less communication between the controller and switches

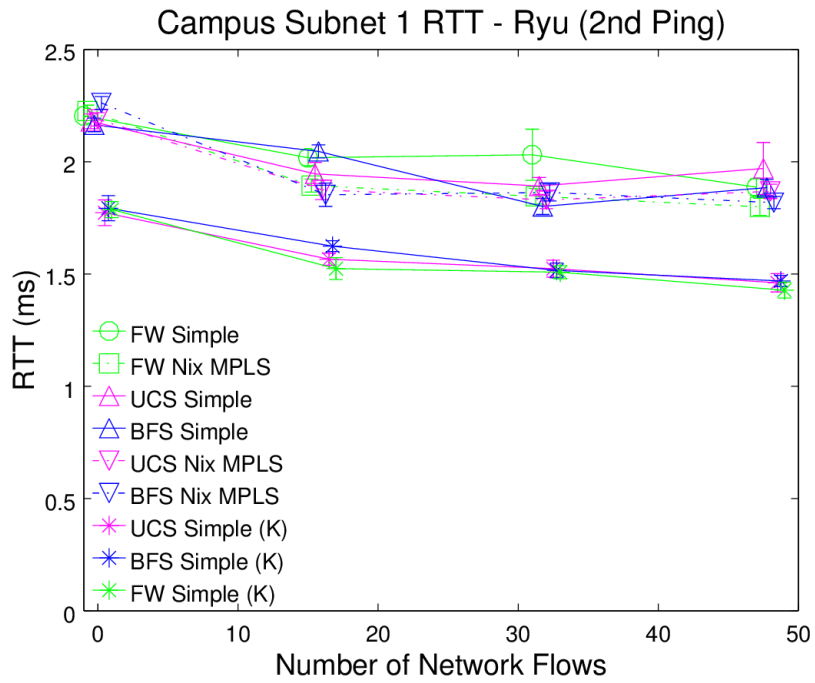


Figure 7.17: Ryu RTT from Subnet 1 (2nd Ping)- Campus Network

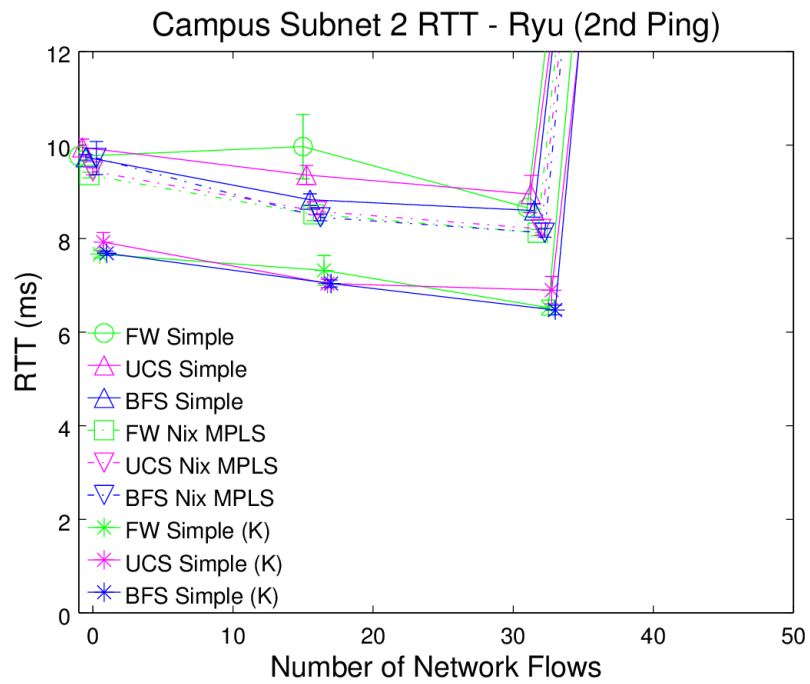


Figure 7.18: Ryu RTT from Subnet 2 (2nd Ping)- Campus Network

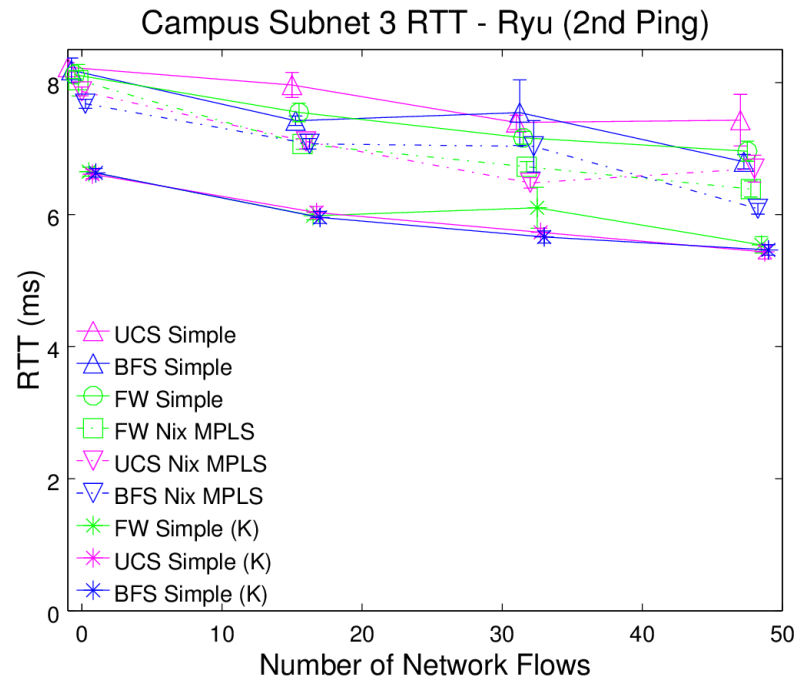


Figure 7.19: Ryu RTT from Subnet 3 (2nd Ping)- Campus Network

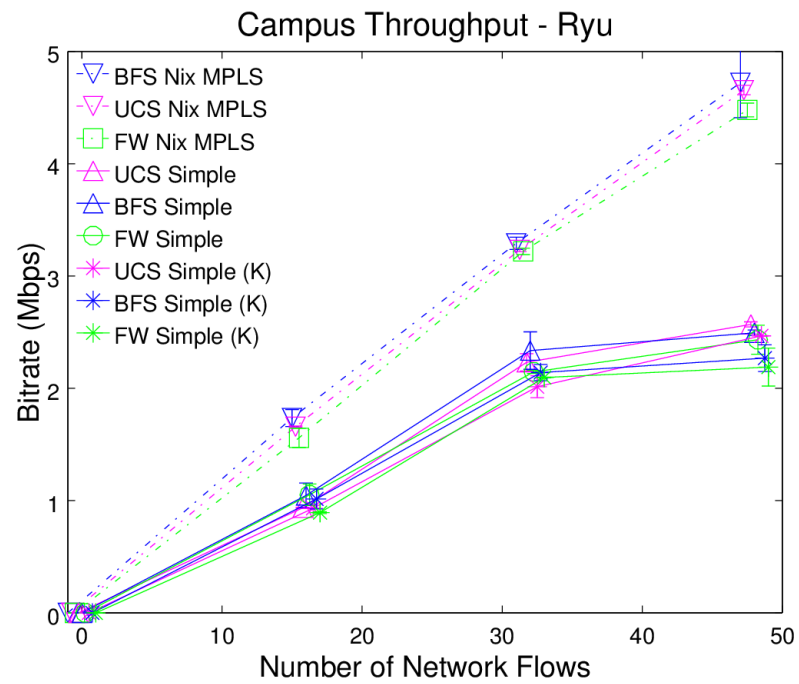


Figure 7.20: Ryu Throughput - Campus Network

alleviates the overall burden on the controller when processing messages.

7.2 GPU-Based Routing with Parallel Floyd-Warshall

Experiments performed on the ring of campus networks have been conducted on the ns-3/DCE simulation framework in conjunction with the native CUDA integration discussed in Section 3.2.1. Generally, function calls relating to time that are executed in DCE will reference the simulated time instead of the wallclock time. A call to the Python `time.time` function, for example, will attempt to call one of the C *time.h* functions, `time` or `gettimeofday`, based on system availability. When executed in DCE, though, these functions will report the simulated time in seconds for the program. Another C function, `clock`, returns the number of processor *clock ticks* which can be used to derive the time in seconds with finer precision than what results from calls to other *time.h* functions. However, processor variability across the numerous brands and architectures of CPUs available presents a modeling issue for DCE that leaves the `clock` method unimplemented in its baseline. As such, the experiments comparing CUDA-based routing to other path algorithms exploit this omission in order to examine the wallclock time rather than the simulated time during the execution of the controller applications under examination. Calls to the Python `time.clock` function return the processor time in seconds based on the value reported by the C `clock` function, which in DCE will simply call the `clock` function natively for these experiments.

Simulation experiments to compare the GPU-based implementation of Floyd-Warshall against other CPU-based path algorithms have been executed on a system with a dual-core 2.8GHz AMD Athlon II X2 220 processor with 4GB of RAM and running Ubuntu 14.04. The GPU used for these experiments is the NVIDIA GeForce GT 730 providing 1GB of memory, 902 MHz GPU clock rate, and 384 CUDA cores. It is a relatively low-cost option compared to some other NVIDIA offerings but provides sufficient power to demonstrate the effectiveness of CUDA-based routing for these experiments. For each

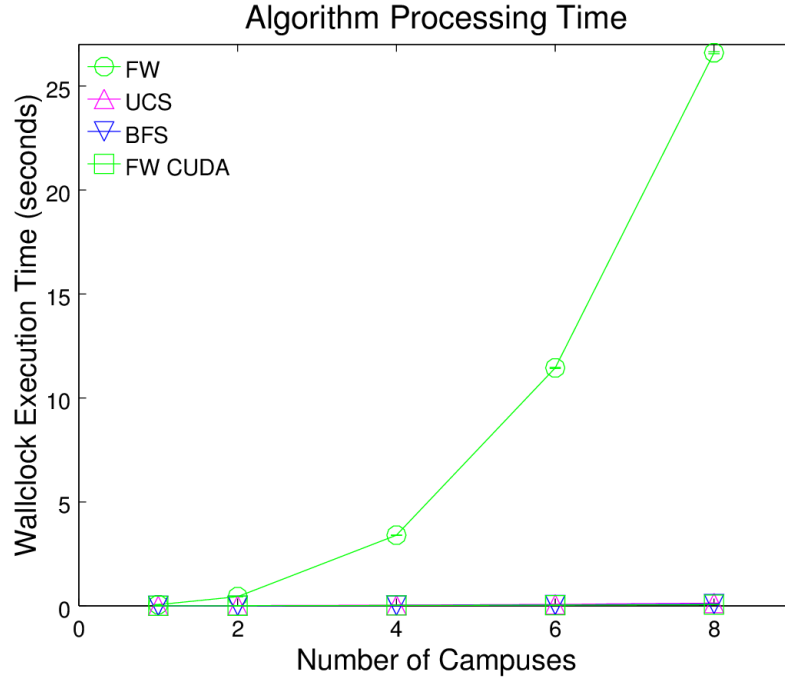


Figure 7.21: Algorithm Processing Time for Ring of Campus Networks

algorithm and network size studied, simulations have been performed 20 times to obtain the average processing time for algorithm computation, path reconstruction, and the total process for each OFPT_PACKET_IN message handled.

As shown in Figures 7.21 and 7.24, simply executing the Floyd-Warshall algorithm sequentially is not a scalable option. For 8 campuses (comprising 144 switches), the algorithm will require nearly 3 million iterations, which as demonstrated, requires over 25 seconds to complete. In contrast, all other times observed, as shown in Figures 7.22, 7.23, and 7.25, occur in milliseconds, which is much more reasonable in a networking environment. Algorithm computation for Floyd-Warshall is significantly reduced when the GPU is allowed to process it. Furthermore, at larger scales, GPU-based Floyd-Warshall outperforms both UCS and BFS. This performance enhancement does not incur a significant cost at the GPU either, never requiring more than 15MB of its memory. Considering the fact that UCS and BFS are only tasked to calculate a single bidirectional path, employing the GPU demonstrates significant improvement by calculating all possible paths in less time.

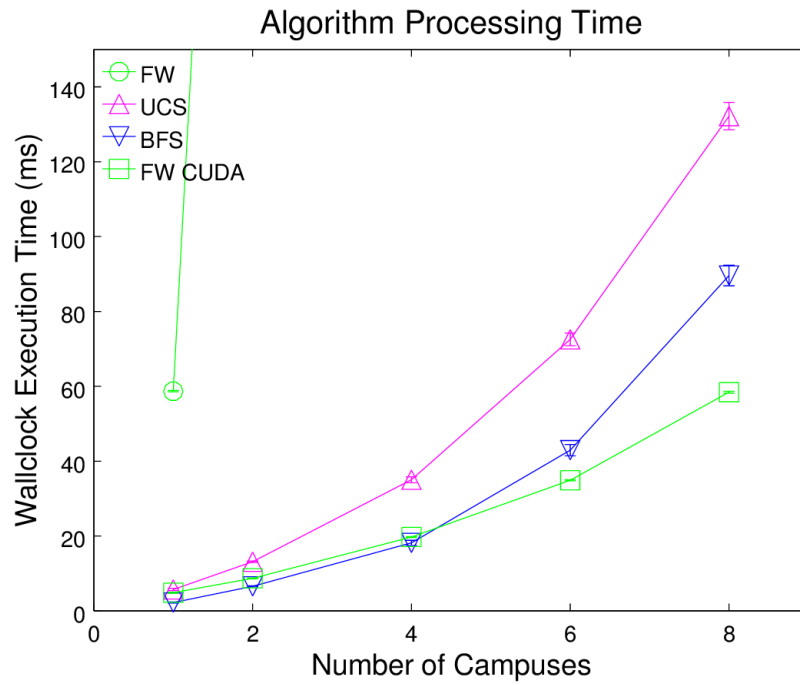


Figure 7.22: Algorithm Processing Time for Ring of Campus Networks (Closer View)

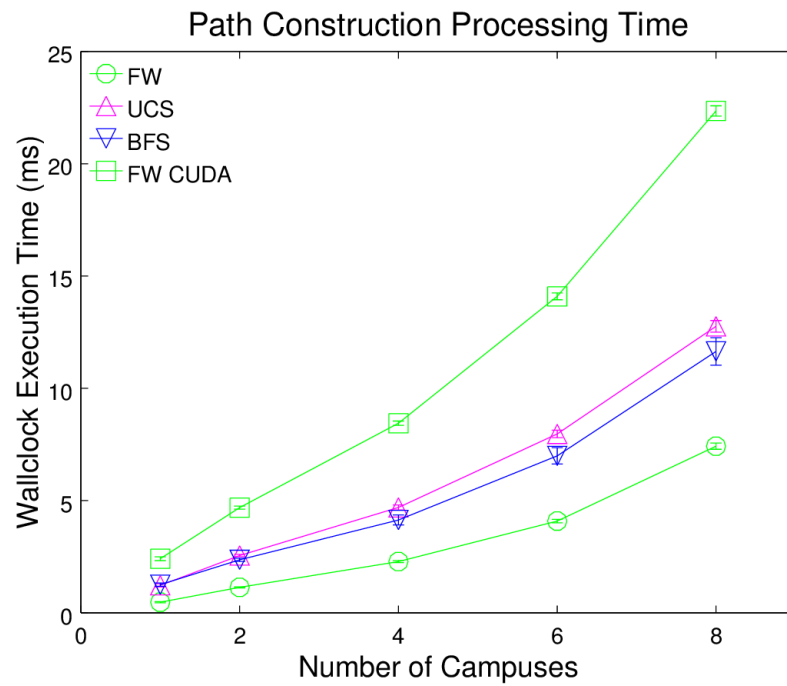


Figure 7.23: Path Construction Processing Time for Ring of Campus Networks

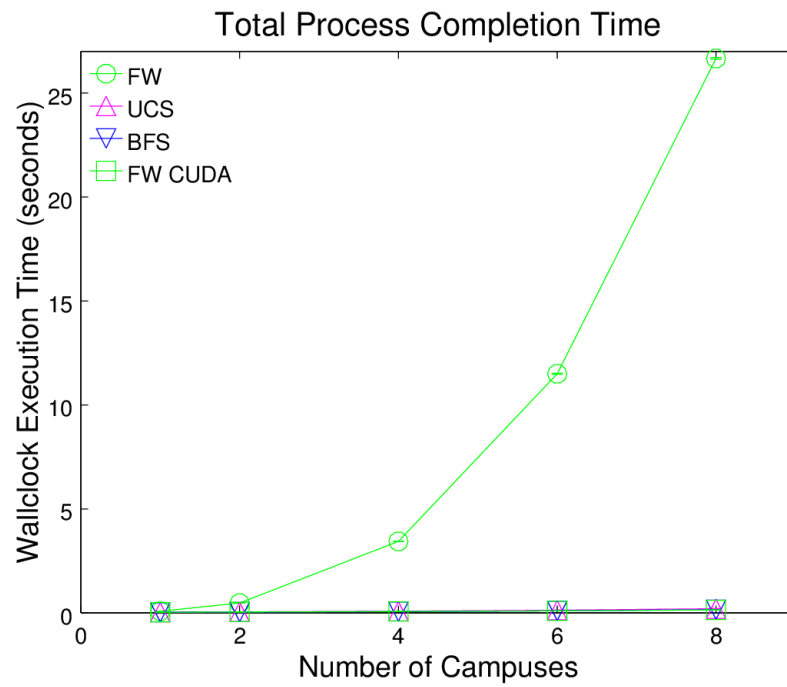


Figure 7.24: Total Process Completion Time for Ring of Campus Networks

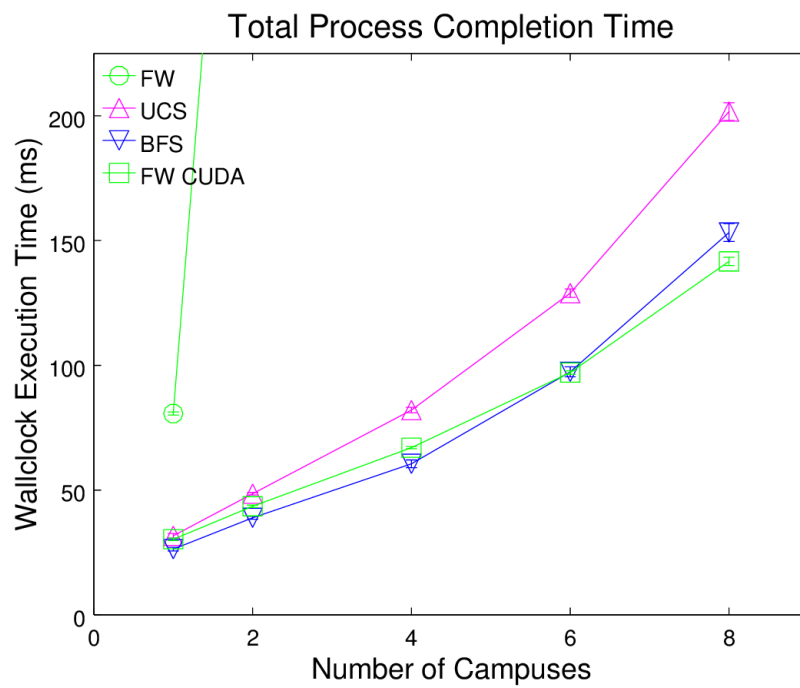


Figure 7.25: Total Process Completion Time for Ring of Campus Networks (Closer View)

This result is still evident for the total processing time required across increasing scales. However, the difference between BFS and the GPU-based routing is less prominent. As shown in Figure 7.23, the main constraint on GPU-based routing is found to be the path reconstruction, a task left entirely to the CPU. Floyd-Warshall without assistance from the GPU is able to provide the lowest path reconstruction time, and this point can be attributed to the relatively straightforward manner that it can traverse the *next* array discussed in Section 5.2. It might be assumed that the GPU-based version of Floyd-Warshall should exhibit similar behavior to its sequential counterpart. However, the PyCUDA library, which relies on the NumPy[69] scientific computing package for Python, requires a number of data type conversions to occur to ensure that numerical comparisons of switch datapath IDs and port numbers occur accurately. These conversions contribute collectively to the overall processing time required to construct the paths from the *next* array. Better methods for handling the path reconstruction may surely be available, and further research will attempt to uncover them to continue to improve the total processing time of GPU-based route calculation.

7.3 Validated Performance Modeling

Experiments performed on the linear and single campus networks have been examined in the ns-3/DCE simulation framework described in Section 4.1. Taking the controller processing times gathered from the experimental results in Section 7.1, multiple simulation models as well as some manipulation schemes of those models have been designed and examined in an effort to determine the most appropriate implementation. The simulated experiments have been executed on a system with a dual-core 2.8GHz AMD Athlon II X2 220 processor running Ubuntu 14.04. This particular system provided 4GB of RAM.

From the experiments performed on GENI aggregates for Section 7.1, controller processing times and the number of primitive Python instructions required to complete processing of particular procedures have been gathered using the Python `time.clock` and

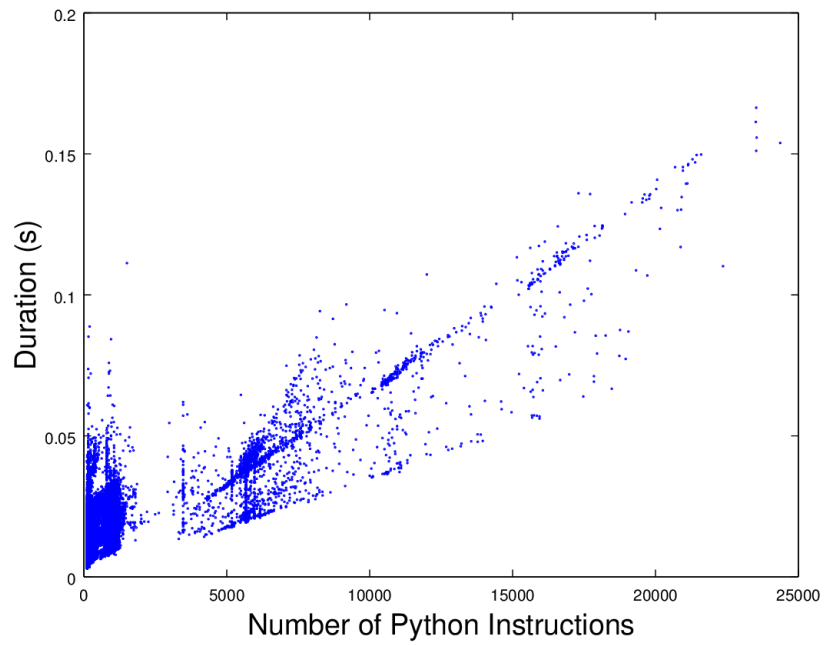


Figure 7.26: Number of Primitive Python Instructions vs. Wallclock Execution Time

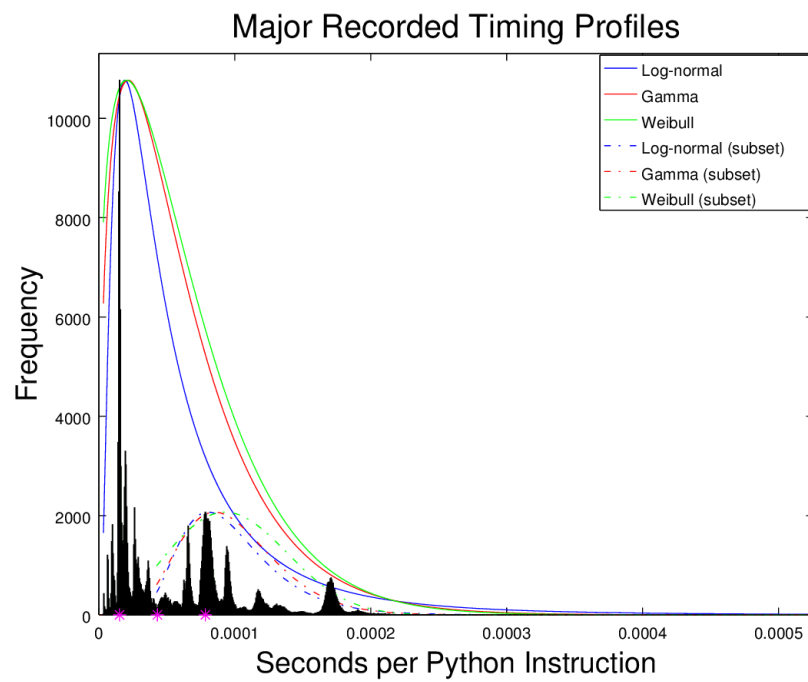


Figure 7.27: Histogram of Python Controller Processing Times

cProfile functionality, respectively. (`time.clock` is replaced with `time.time` in simulation.) These results are shown in Figure 7.26. From this data, a crude measure of the time required to complete Python instructions has been calculated, as noted by the “Seconds per Python Instruction” displayed in the histogram in Figure 7.27. These values will be referred to as the *instruction processing time* (T_{ip}). Certain assumptions are made when considering the data. Network programs, whether they are as simple as an application like *ping* or as complex as those used by SDN controllers, generally will only make significant use of a subset of a programming language and its libraries. Furthermore, as noted in Section 5.3, composing timing models based on the intricate nuances of various CPU architectures and underlying instruction sets is unnecessarily tedious in conjunction with the current design of DCE. In attempting to simplify the resulting model, the notion that only a subset of a programming library will see heavy usage limits the complexity required by the model. It also suggests that these more heavily used components will produce timing patterns that can be observed which generate histograms that can subsequently be used to fit a simulated model.

An additional assumption is made regarding the nature of Python instructions compared to the *glibc* functions for which DCE is designed. Because it is already being assumed that only a subset of the Python libraries will see significant usage, a similar conjecture can be made regarding the utilization of *glibc* for the underlying C language. The timing patterns more so than the specific time measurements of distinct functions are of importance for the simulated model. Therefore, the transitive notion that the patterns evoked for Python instructions would produce similar trends for the C functions on which Python is ultimately executing is assumed when examining the models.

Although controllers are operating continuously within an SDN, specific event handling functions prompted for a particular controller application can be monitored to gather timing and profile data. For the applications studied in Section 7.1, these functions are elicited based on `OFPT_MULTIPART_REQUEST` and `OFPT_PACKET_IN` messages. The

OFPT_MULTIPART_REQUEST messages are received immediately after the controller and a switch have completed the OpenFlow handshake. For NIX-MPLS flow rule installation, these messages are used by the controller to determine when a switch is in the correct state to receive instructions for OFPAT_POP_MPLS actions. The OFPT_PACKET_IN messages cause the controller applications to instruct switches for ARP handling or packet routing. A number of other operations occur as well over the course of a controller timeline. However, these procedures, such as LLDP packet handling, host discovery, and topology data requests occur in negligible durations compared to the aforementioned operations, are primarily used for accessing and storing information, and do not interact with any Python primitive instructions which would call *glibc* functions. For these reasons, the operations concerning OFPT_MULTIPART_REQUEST and OFPT_PACKET_IN messages are considered the “major” ones regarding this modeling effort.

The data shown in Figure 7.27 represents frequency distribution for the instruction processing times for “major” operations for all controller testing in GENI. From this data, three probability distributions are fitted to the data and examined: gamma, log-normal, and Weibull. These distributions have been primarily selected based on their immediate visual resemblance to the data as well as their baseline support provided in ns-3. The solid lines from the figure represent the distributions fitted based on the entirety of the dataset. As noted by the first asterisk in the figure, a significant peak occurs for approximately $15\mu\text{s}$. It is evident from the fitted lines that this particular peak significantly influences the resulting distribution models. The second and third asterisks in the figure note a local minimum at $43\mu\text{s}$ and a local maximum at $78\mu\text{s}$ that serve as additional features under consideration when evaluating the examined simulation models. The set of dashed lines in the figure fit a subset of the original data beginning with time values at the $43\mu\text{s}$ local minimum to the maximum time value observed. This subset of the data, as noted in Table 7.4, represents just over half of the total samples collected from experimentation in GENI. However, as Table 7.3 points out, these samples contribute 85% of the observed timing influence, proving

to be a key contributor when evaluating a simulated model.

Random variable streams are introduced for event scheduling in the *task manager* of DCE. These variable streams take the fitting parameters calculated from the entirety of the controller processing time dataset. Informal initial testing demonstrated a need to scale the values produced by the random variable streams in order to more closely resemble the original data. For this reason, a scaling factor of 100 is considered. Initial histogram results are shown in Figures 7.28, 7.29, and 7.30. For each controller application running the first three of the four traffic generation patterns from Section 6.2, 10 simulations have been performed, each using the same seed but a different seed run for the random number generator[70]. The general behaviors of the initial models produce some resemblance to the original data. Testing over the course of this effort presented the limitation that DCE would not mimic the $15\mu s$ global maximum but could still adequately produce results that fit the three mentioned features and the subset model. To elaborate, by using the fitted distributions for the entire original dataset, the features and the subset patterns could be produced. However, the span over which these resultant distributions occur does not effectively map to either the original model or the subset model. Results from using the log-normal distribution provide a minor visual resemblance, but it is still not considered sufficient.

Secondary testing, shown in Figures 7.31, 7.32, and 7.33, has implemented antithetic sampling for the distributions of the random variable streams. Again, for each controller application running the first three of the four traffic generation patterns from Section 6.2, 10 simulations have been performed, each using the same seed but a different seed run. Antithetic sampling provides a statistical mechanism by which variance in a sample population can be reduced, and this variance reduction is evident in testing through the “tightening” of the histogram data seen in the figures. This tightening produces models that more closely match the features and subset models of the original data.

Further manipulation is necessary in order to improve the models. Transformational

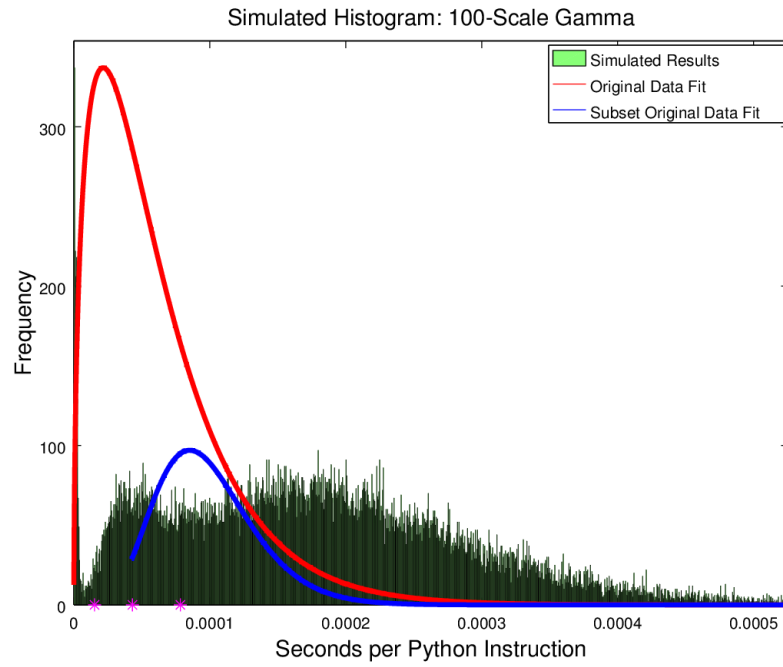


Figure 7.28: Histogram for gamma fitting
with $\alpha=1.53398$ and $\beta=4.03046e-5$ whose sampled values are scaled by a factor of 100.

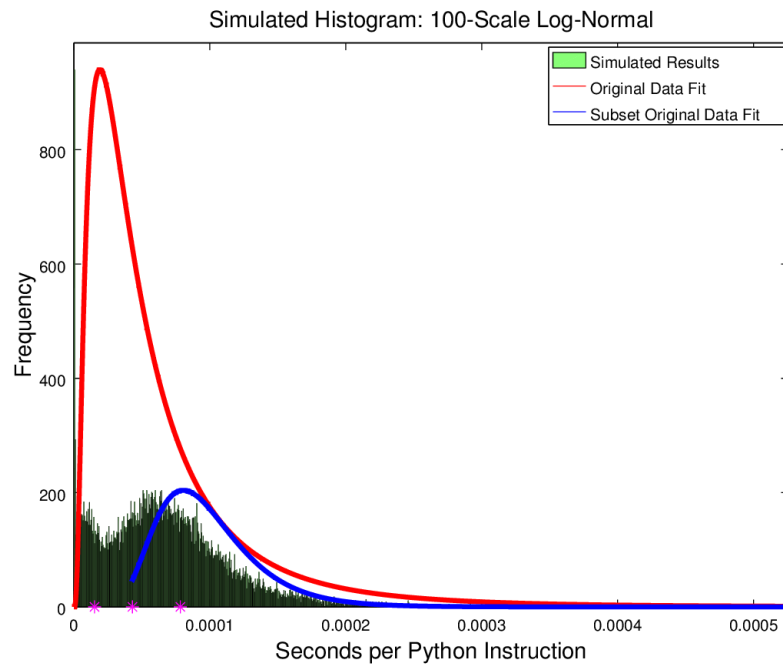


Figure 7.29: Histogram for log-normal fitting
with $\mu=-10.05126$ and $\sigma=0.89957$ whose sampled values are scaled by a factor of 100.

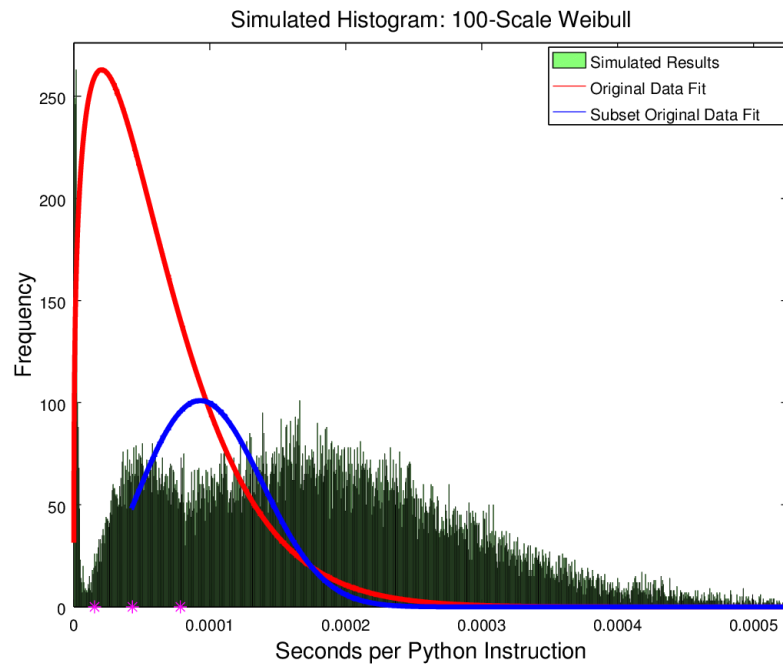


Figure 7.30: Histogram for Weibull fitting with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values are scaled by a factor of 100.

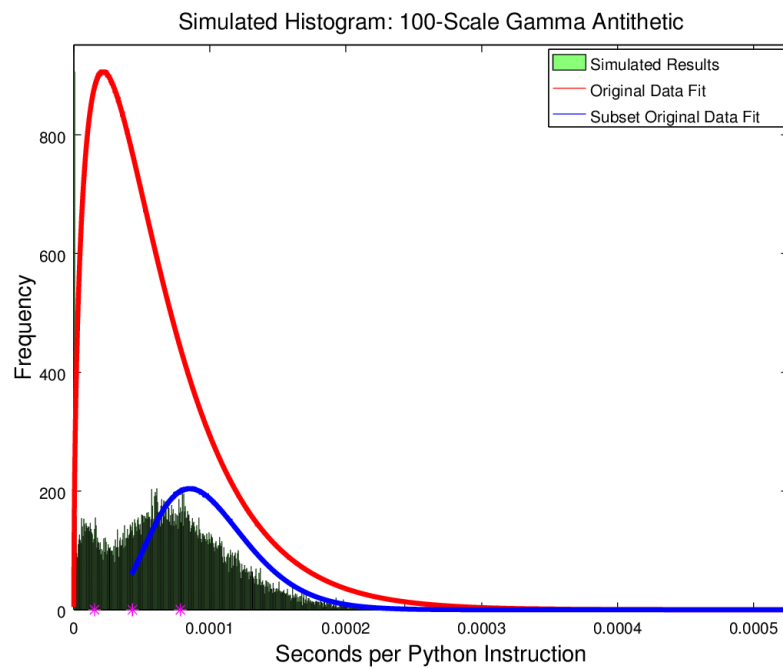


Figure 7.31: Histogram for gamma fitting with antithetic sampling with $\alpha=1.53398$ and $\beta=4.03046e-5$ whose sampled values are scaled by a factor of 100.

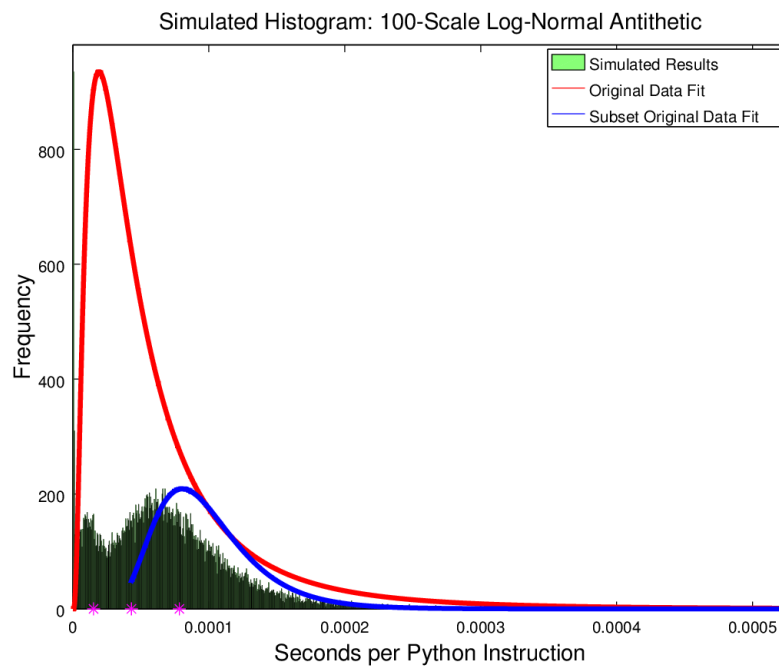


Figure 7.32: Histogram for log-normal fitting with antithetic sampling with $\mu=-10.05126$ and $\sigma=0.89957$ whose sampled values are scaled by a factor of 100.

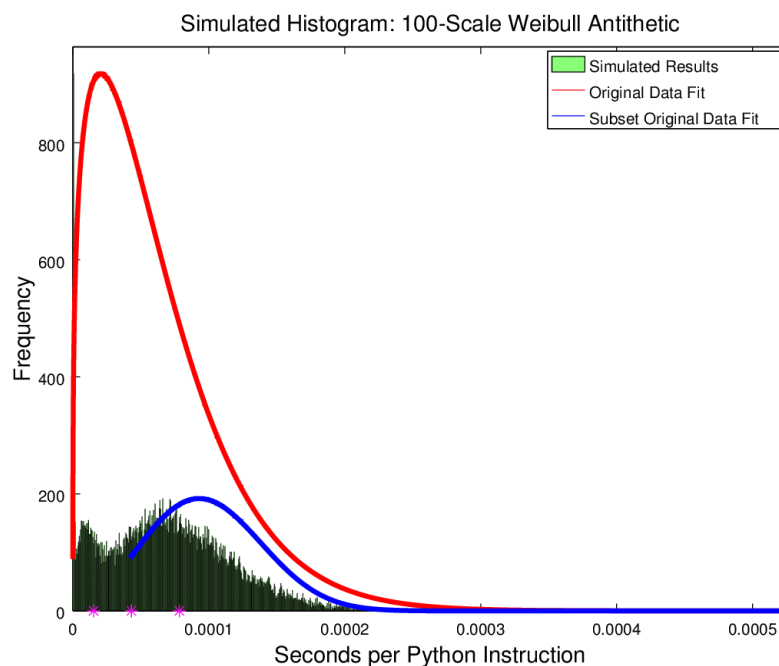


Figure 7.33: Histogram for Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values are scaled by a factor of 100.

shifting of the random variable streams is one potential avenue for moving the resulting distributions toward both the features and the subset model. Indeed, instruction processing times less than $3\mu s$ are not observed in the real-world original data since no real-time processing can possibly occur in zero-time. This $3\mu s$ shift though would still not be sufficient based on differences between the original data features and their simulated analogues. An additional caveat is the necessity to provide some bound to the maximum T_{ip} allowed to both maintain realism and ensure that scheduled times do not cause the simulated network to stall. In this way, the Weibull distribution in the ns-3 baseline provides the most thorough pathway for continued adjustments to the model. For the Weibull distribution results shown in Figure 7.33, the difference between the original features and their simulated analogues is $12.09\mu s$, and this value is used to shift the random variable samples prior to scaling. The maximum recorded T_{ip} from the original data is $522.687\mu s$, and this value is used as a simple bound on the values permitted for the Weibull distribution. For both models and for each controller application running the first three of the four traffic generation patterns from Section 6.2, 20 simulations have been performed, each using the same seed but a different seed run.

The histogram results for this shifting and bounding are shown in Figures 7.34 and 7.35. Figure 7.35 is simply provided to visualize the low occurrence of T_{ip} values beyond the originally recorded maximum. (Additionally, all raw controller processing data is presented in Appendix A to visually convey how the simulated results compare to the data in Figure 7.26.) Figure 7.34 visually demonstrates the improved nature of the simulated model to fit the noted features and the subset model. Tables 7.1, 7.2, 7.3, 7.4, and 7.5 aim to provide a better understanding of how all of the proposed models compare. In Table 7.1, the bound Weibull distribution provides the lowest time and T_{ip} as well as the second highest maximum number of instructions, i.e. the second closest to the originally recorded maximum. In Table 7.2, the root sum of squares (RSS) is calculated for each model based on the observed distances of the simulated minimum and maximum features from those in

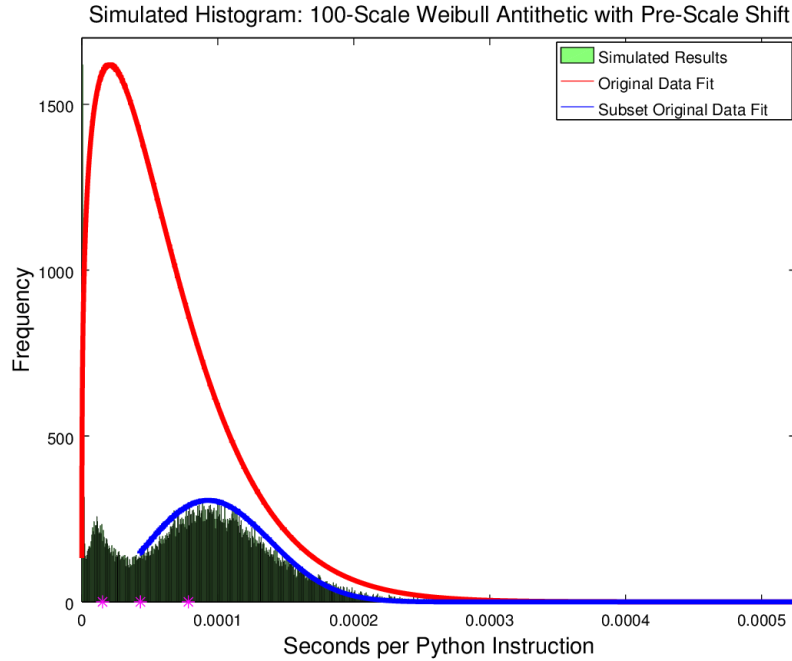


Figure 7.34: Histogram for bound Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values (with a maximum of $522.687\mu s$) are shifted by $12.09\mu s$, then scaled by a factor of 100.

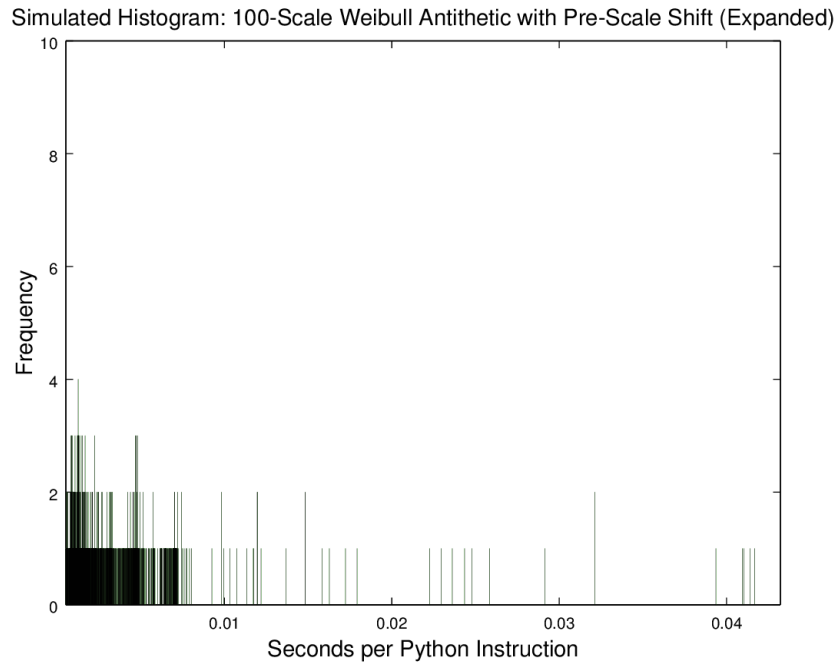


Figure 7.35: Remaining histogram for bound Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values (with a maximum of $522.687\mu s$) are shifted by $12.09\mu s$, then scaled by a factor of 100.

the original data. With an RSS of 14.104, the bound Weibull distribution shows the lowest variation from the originally recorded features. Tables 7.3 and 7.4 are presented to numerically distinguish the quality of each model as it pertains to two different methods for examining the composition of each dataset. In Table 7.3, data is represented based on the amount of time that portions of the histograms comprise. In this way, lower percentages beyond the maximum T_{ip} are preferable. Also, while the extent of the global maximum exhibited by the original data is difficult to replicate, closer proportions of the other two percentage columns to the original data results in Table 7.3 would be a goal when further improving the model. Table 7.4 presents data based on the number of samples that compose each dataset. Similar reasoning to that for Table 7.3 can be used when examining this table. The bound Weibull distribution provides the third lowest percentage of time outside the originally recorded maximum and the lowest percentage of samples in this range. Both the regular and antithetic sampling for the log-normal distributions provide the most reasonable compositions for the other percentages considered. Future research would benefit from examining additional manipulation schemes on this particular distribution.

Table 7.5 provides a statistical analysis of each proposed distribution model to better gauge the similarity of the simulated models to the original data. In the case of the Mann-Whitney U test, also referred to as a *rank-sum* test, the null hypothesis states that when selecting a value from each of two sample populations, it is equally likely that one value is greater or less than the other value. For the Kolmogorov-Smirnov test, the null hypothesis examines whether two sample populations could have been derived from the same probability distribution. These statistical tests are used to compare a subset of the original data, beginning at the $43\mu s$ local minimum and extending to the maximum T_{ip} observed, against each simulated dataset using the same range constraints. For all tested models, it is difficult to statistically claim that they are similar to the original data. The nearest approach to a statistically similar model is the Weibull distribution with antithetic sampling and no bounding based on the proximity of its statistics to zero. In fact, the result of shifting and

bounding this distribution is actually less statistical similarity comparatively. However, the analytical data from the other tables can still be used to suggest that statistical similarity can be sacrificed for a more practically similar model in this case. Due to the simplifying assumptions originally considered, statistical similarity is not necessarily a requirement but instead a goal for which a stricter or more robust simulation model should strive. Although not statistically similar, the ability for the proposed simulation model, in this particular case the bound Weibull distribution with antithetic sampling, to capture some of the more prominent aspects of the original data provides a sufficient mechanism for adequately modeling the T_{ip} behaviors in DCE.

Table 7.1: Simple sampling results of modeling validation experiments

		Parameters	Total Samples	Maximum Instructions	Maximum Time (s)	Maximum T_{ip} (s/Instruction)
Original Data	Gamma	$\alpha=1.53398$ $\beta=4.03046e-4$	191702	24364	0.1664	5.23e-4
	Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	40805	14599	7.7570	0.0535
	Weibull	$shape=6.69475e-5$ $scale=1.28053$	37776	16843	7.0691	0.0388
	Gamma	$\alpha=1.53398$ $\beta=4.03046e-4$	39574	17307	9.0180	0.0752
Antithetic Sampling	Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	36985	16849	10.0735	0.0839
	Weibull	$shape=6.69475e-5$ $scale=1.28053$	37134	16884	6.6545	0.0416
	Weibull Bound	$shape=6.69475e-5$ $scale=1.28053$ $shift=1.209e-5$ $bound=5.22687e-4$	36185	26264	9.7635	0.0579
	Weibull Bound		66810	20249	5.1861	0.0432
Regular Sampling						

Table 7.2: Feature-based results of modeling validation experiments
(Time values are reported in μs)

	Parameters	Regular Sampling						Antithetic Sampling					
		Local Max 1	Difference from Local Max 1	Local Min	Difference from Local Min	Local Max 2	Difference from Local Max 2	Root Sum of Squares					
Original Data		15.244		43.104		78.322							
Gamma	$\alpha=1.53398$ $\beta=4.03046e-4$	50.988	35.744	105.655	62.552	179.772	101.449	124.428					
Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	9.462	-5.782	33.642	-9.462	53.616	-24.706	27.081					
Weibull	$shape=6.69475e-5$ $scale=1.28053$	50.462	35.218	82.001	38.898	166.105	87.783	102.270					
Gamma	$\alpha=1.53398$ $\beta=4.03046e-4$	15.769	0.525	29.962	-13.142	60.975	-17.347	24.907					
Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	15.244	0.000	25.231	-17.872	60.975	-17.347	24.907					
Weibull	$shape=6.69475e-5$ $scale=1.28053$	11.564	-3.680	31.013	-12.090	66.232	-12.091	17.490					
Weibull Bound	$shape=6.69475e-5$ $scale=1.28053$ $shift=1.209e-5$ $bound=5.22687e-4$	11.039	-4.205	34.693	-8.411	88.835	10.512	14.104					

Table 7.3: Time-based results of modeling validation experiments

	Parameters	$\% \in [0, \text{OriginalLocalMin})$	$\% \in [\text{OriginalLocalMin}, \text{OriginalMaxT}_{ip})$	$\% \in [\text{OriginalMaxT}_{ip}, \infty)$
Regular Sampling	Original Data	14.95	85.05	0.00
	Gamma	$\alpha=1.53398$ $\beta=4.03046\text{e-}4$	98.65	0.11
	Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	90.65	1.81
	Weibull	$shape=6.69475\text{e-}5$ $scale=1.28053$	98.76	0.09
Antithetic Sampling	Gamma	$\alpha=1.53398$ $\beta=4.03046\text{e-}4$	91.95	1.59
	Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	90.99	1.80
	Weibull	$shape=6.69475\text{e-}5$ $scale=1.28053$	91.89	1.54
	Weibull Bound	$shape=6.69475\text{e-}5$ $scale=1.28053$ $shift=1.209\text{e-}5$ $bound=5.22687\text{e-}4$	95.47	0.49

Table 7.4: Sample-based results of modeling validation experiments

	Parameters	$\% \in [0, \text{OriginalLocalMin})$	$\% \in [\text{OriginalLocalMin}, \text{OriginalMaxT}_{ip})$	$\% \in [\text{OriginalMaxT}_{ip}, \infty)$
		46.74	53.26	0.00
Regular Sampling	Original Data			
	Gamma	$\alpha=1.53398$ $\beta=4.03046\text{e-}4$	86.34	2.80
	Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	65.75	3.02
	Weibull	$shape=6.69475\text{e-}5$ $scale=1.28053$	87.32	2.52
Antithetic Sampling	Gamma	$\alpha=1.53398$ $\beta=4.03046\text{e-}4$	68.19	3.21
	Log Normal	$\mu=-10.05126$ $\sigma=0.89957$	66.44	3.16
	Weibull	$shape=6.69475\text{e-}5$ $scale=1.28053$	67.69	3.04
	Weibull Bound	$shape=6.69475\text{e-}5$ $scale=1.28053$ $shift=1.209\text{e-}5$ $bound=5.22687\text{e-}4$	75.37	1.96

Table 7.5: Statistical similarity results of modeling validation experiments

	Parameters	Mann-Whitney			Kolmogorov-Smirnov		
		p	z	p	ks	d	
Original Data		1	0	1	0	0	
Gamma	α =1.53398 β =4.03046e-4	0	173.92	0	84.25	0.52	
Log Normal	μ =-10.05126 σ =0.89957	0	-36.61	0	26.83	0.19	
Weibull	$shape$ =6.69475e-5 $scale$ =1.28053	0	169.84	0	82.84	0.52	
Gamma	α =1.53398 β =4.03046e-4	0	-23.69	0	20.71	0.15	
Log Normal	μ =-10.05126 σ =0.89957	0	-35.68	0	26.86	0.19	
Weibull	$shape$ =6.69475e-5 $scale$ =1.28053	0	-19.79	0	19.67	0.14	
Weibull Bound	$shape$ =6.69475e-5 $scale$ =1.28053 $shift$ =1.209e-5 $bound$ =5.22687e-4	0	50.74	0	41.14	0.22	

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This work has covered a variety of concepts and experimental research in terms of prototyping improvements for software-defined networks and providing more realistic models for examining them within simulation. The originally proposed NIX vector construct for dynamic, on-demand network routing has been borrowed to develop the NIX-MPLS flow rule installation behavior for SDN. By deploying NIX-MPLS flow rule installation in certain SDN controller libraries, switches can be instructed in a quicker, more succinct manner that, based on the observed results, suggests an improved effective throughput in the SDN. Path determination utilizing the massive parallelism of GPUs has also been studied, demonstrating improved processing time when using the Floyd-Warshall algorithm. Furthermore, the parallelized version of the Floyd-Warshall algorithm using the GPU provides faster processing times when calculating all paths in an SDN to simpler single-path computation using sequential versions of BFS and UCS. Finally, a model has been designed and evaluated for an adequate representation of instruction processing time distributions in an SDN controller operating in simulation. The approach and pathway to achieving this model has been described, ultimately deeming a shifted and bound Weibull distribution with antithetic sampling as an appropriate approximation. This distribution has been introduced into the task scheduling mechanisms of DCE to influence a random variable stream responsible for providing time values to the event scheduler. These values in turn permit the network simulation to advance its simulation time in a manner that resembles realistic processing times for the SDN controller.

Future research will focus on some of the current issues related to the implementation of the research and certain shortcomings of the experimental setups. Although NIX-MPLS behavior provided some improvements over typical flow rule installation, it is constrained

in its use of MPLS headers. An improvement that fully deploys a NIX vector header that can be understood by both the controller and its switches would benefit from not only the improvements toward flow rule installation but also a more concise representation of the NIX vector within its own specified header. This header would require its own EtherType that the controller and switches understand and include the NIX vector itself and values representing the original EtherType of the packet, the total length of the NIX vector, and its current length in relation to the location of the packet in the path. The switches would then require experimental flow rules that would allow them to parse and modify the NIX vector and appropriately forward the packets based on it. GPU-based packet routing for SDN, although initially promising based on the results of this research, still requires testing within a more adequate testbed. The sequential nature of event scheduling in ns-3 does not permit a true tasking of the GPU on the controller in the presence of typical network traffic. A more appropriate real-time testbed would allow a better representation of packet reception and handling within the network to better determine how the GPU may potentially cause a bottleneck under more significant network stress. Additionally, further testing on improvements to the path reconstruction algorithms utilized in conjunction with the parallelized Floyd-Warshall algorithm can provide more significant reductions in overall processing times.

Although the simulation model determined for mimicking the controller processing times produced an adequate representation, further testing could continue to determine its appropriateness on a broader scale. The model presented can be most specifically considered as a representation in network simulation of results observed in GENI. Greater variety in terms of the processors and environments used for hardware real-time test results could be examined, and based on these results, the applicability of the current model could be either corroborated or adjusted accordingly. Validating the performance of the simulated model across as many variables and for as many scenarios as possible improves its realism and provides further justification in support of its usage.

Appendices

APPENDIX A

CONTROLLER PROCESSING TIME DATA

The figures provided in this appendix show the raw controller processing data for each of the simulated models discussed in Section 7.3. Graphs are presented with the number of primitive Python instructions gathered from the `cProfile` instruction profiling capability in Python against the time required to complete a particular number of instructions. For each case, one figure presents the entire data set such that it can be visualized and understood in its complete context. In a second figure for each simulation model, each figure is limited on the x- and y- axes to the maximum values derived from the original data set gathered through experiments run on GENI aggregates. These figures provide a better view of the majority of the simulated data results within the constraints of the original dataset limits.

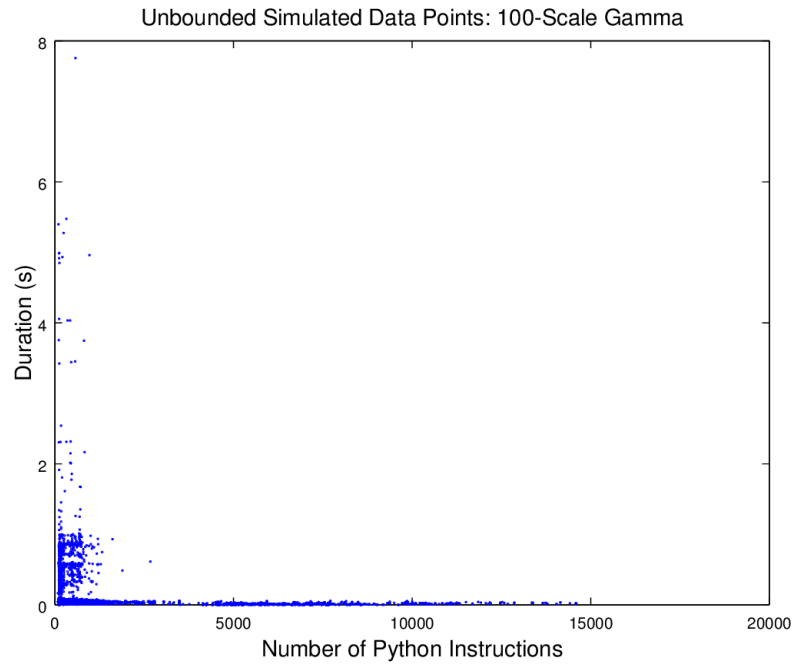


Figure A.1: Total simulated data for gamma fitting with $\alpha=1.53398$ and $\beta=4.03046\text{e-}5$ whose sampled values are scaled by a factor of 100.

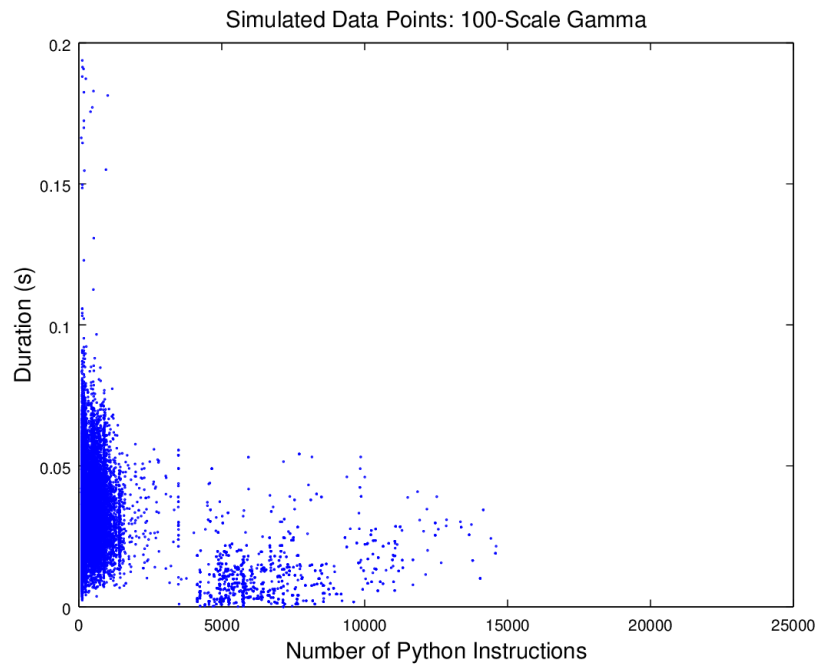


Figure A.2: Subset of simulated data for gamma fitting with $\alpha=1.53398$ and $\beta=4.03046\text{e-}5$ whose sampled values are scaled by a factor of 100.

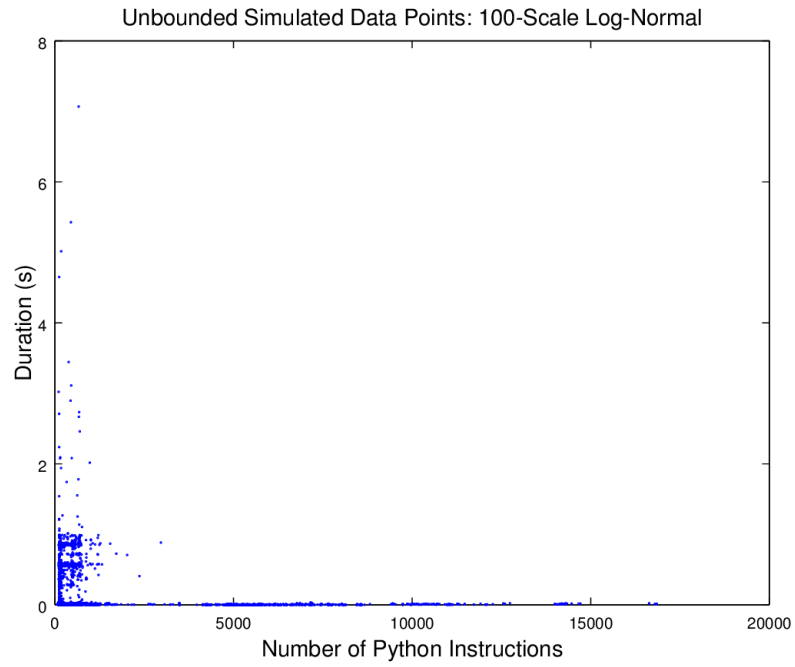


Figure A.3: Total simulated data for log-normal fitting with $\mu=-10.05126$ and $\sigma=0.89957$ whose sampled values are scaled by a factor of 100.

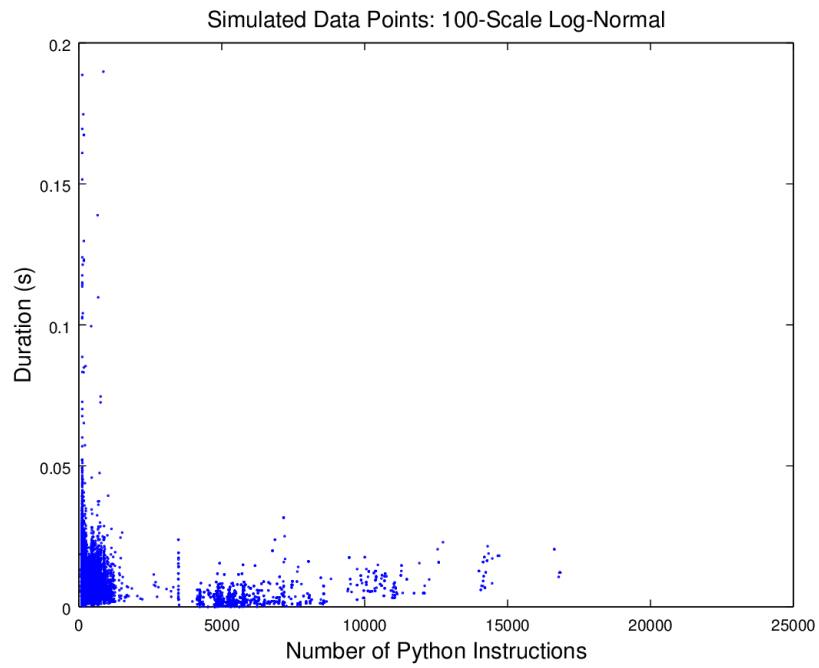


Figure A.4: Subset of simulated data for log-normal fitting with $\mu=-10.05126$ and $\sigma=0.89957$ whose sampled values are scaled by a factor of 100.

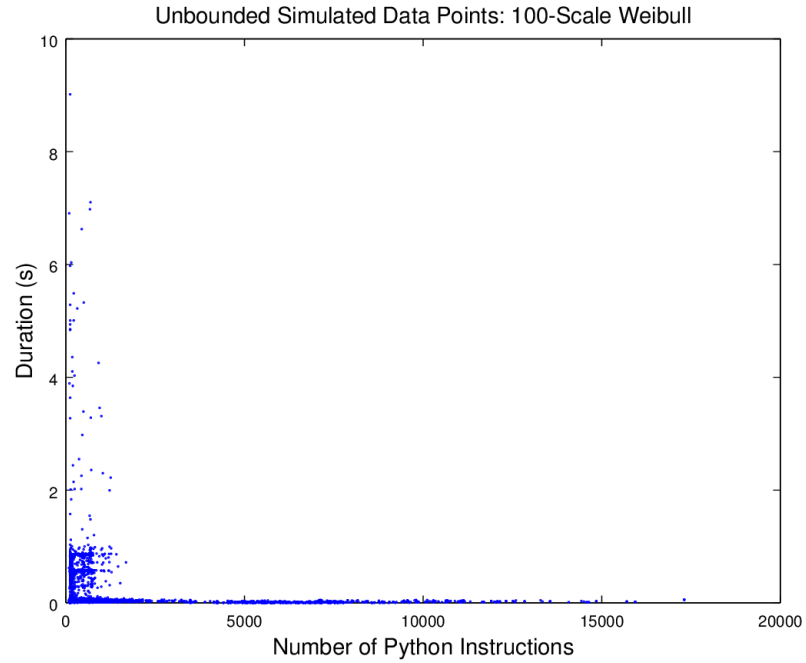


Figure A.5: Total simulated data for Weibull fitting with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values are scaled by a factor of 100.

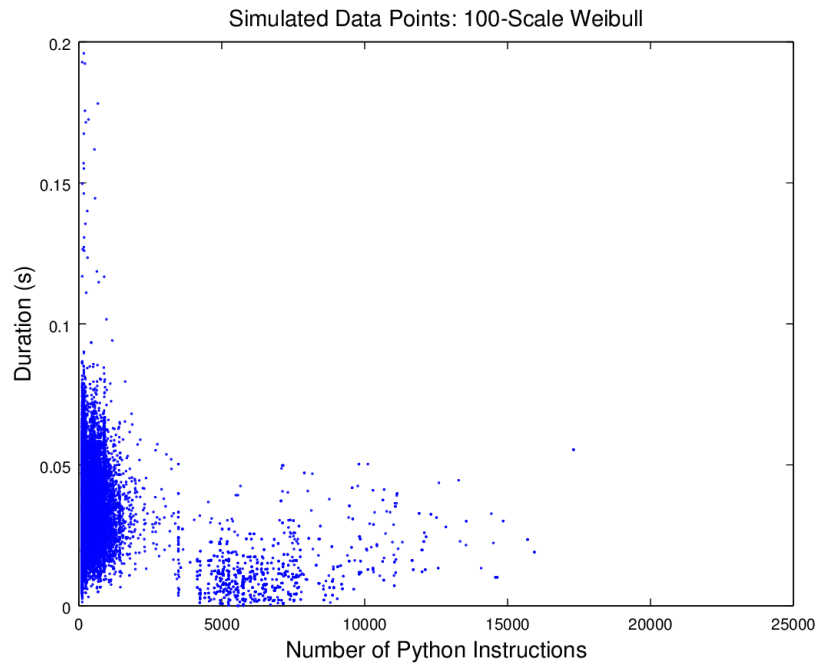


Figure A.6: Subset of simulated data for Weibull fitting with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values are scaled by a factor of 100.

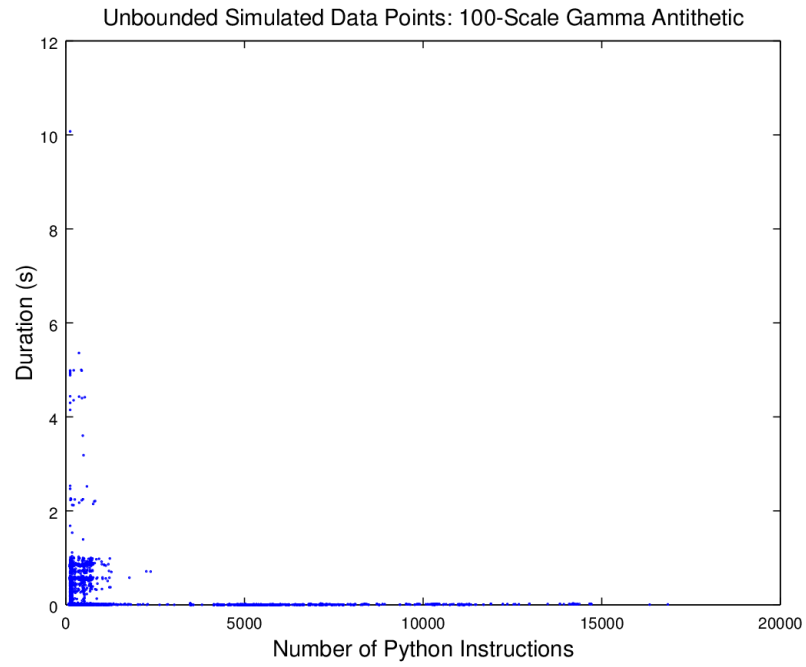


Figure A.7: Total simulated data for gamma fitting with antithetic sampling with $\alpha=1.53398$ and $\beta=4.03046e-5$ whose sampled values are scaled by a factor of 100.

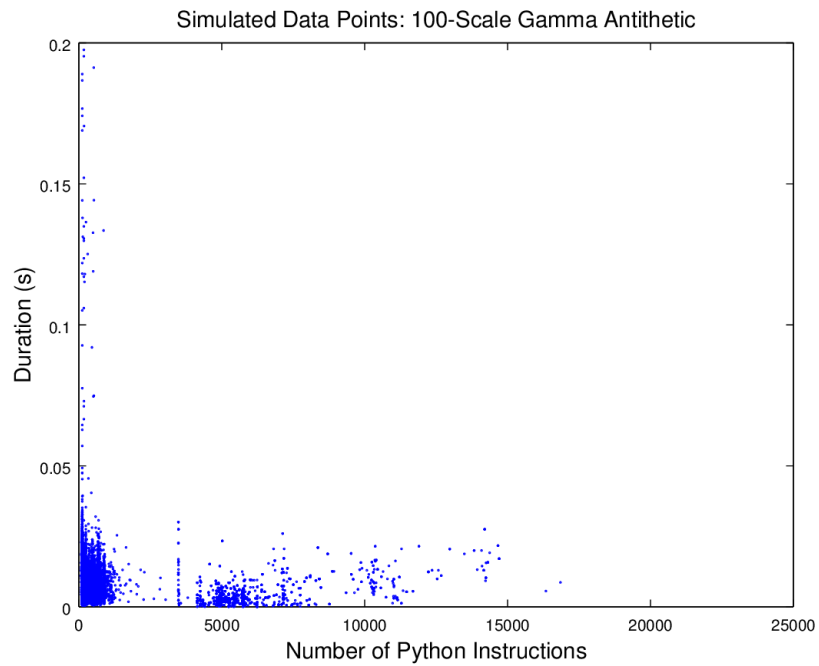


Figure A.8: Subset of simulated data for gamma fitting with antithetic sampling with $\alpha=1.53398$ and $\beta=4.03046e-5$ whose sampled values are scaled by a factor of 100.

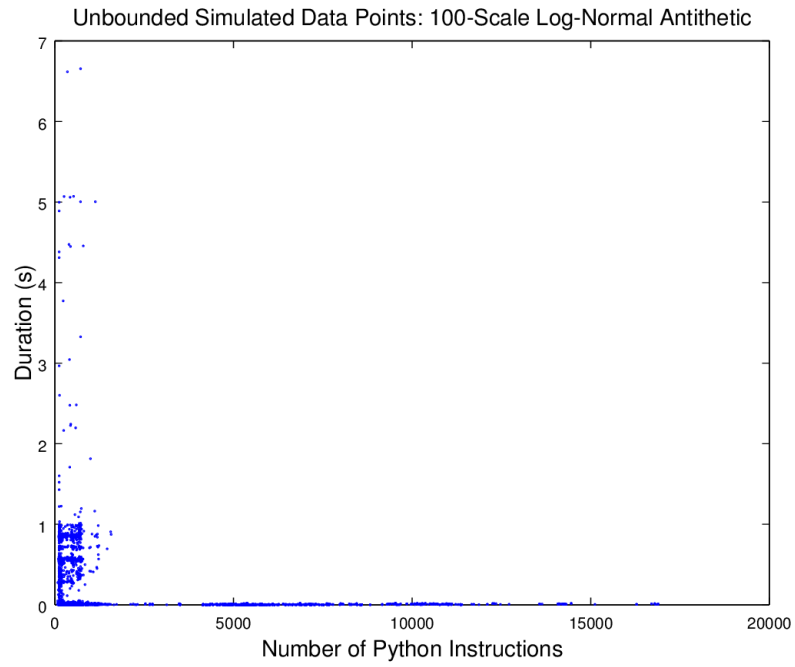


Figure A.9: Total simulated data for log-normal fitting with antithetic sampling with $\mu=-10.05126$ and $\sigma=0.89957$ whose sampled values are scaled by a factor of 100.

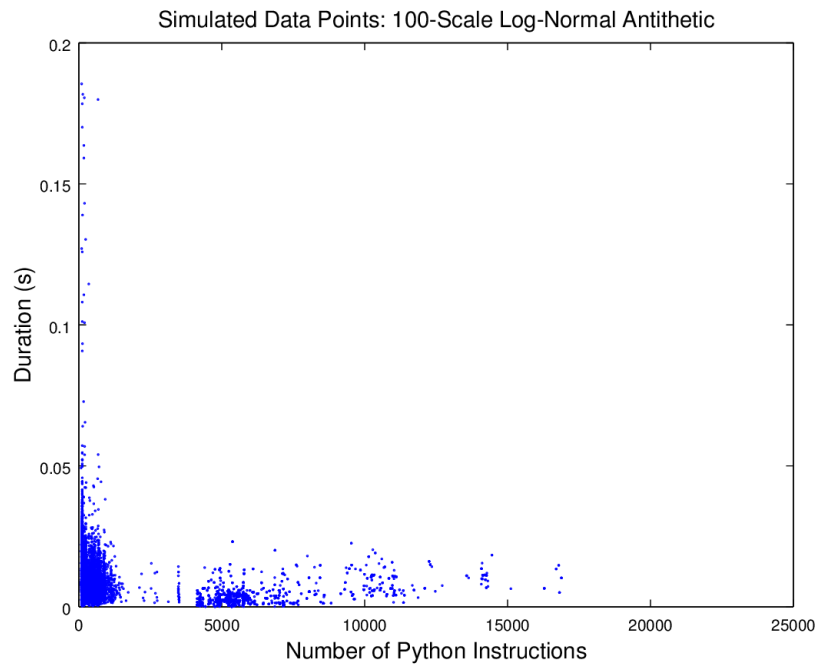


Figure A.10: Subset of simulated data for log-normal fitting with antithetic sampling with $\mu=-10.05126$ and $\sigma=0.89957$ whose sampled values are scaled by a factor of 100.

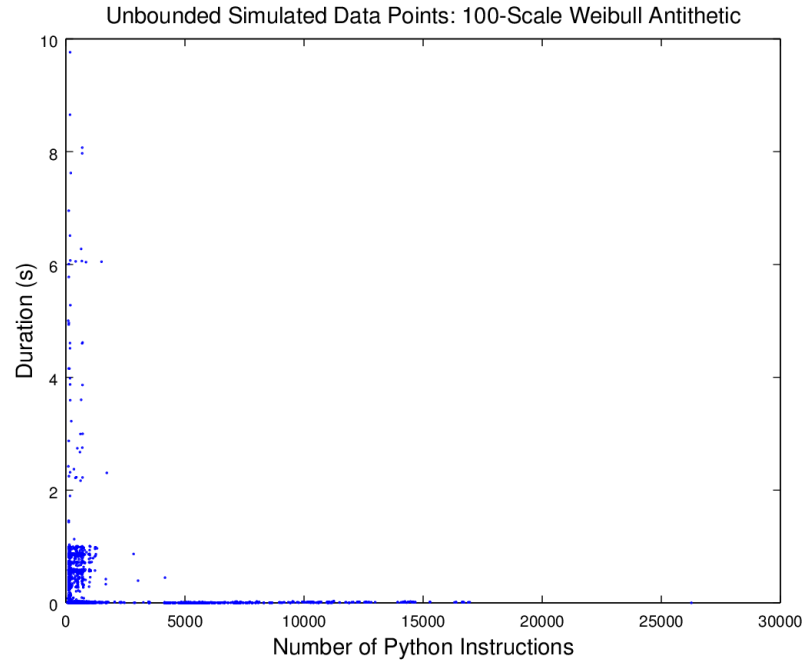


Figure A.11: Total simulated data for Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values are scaled by a factor of 100.

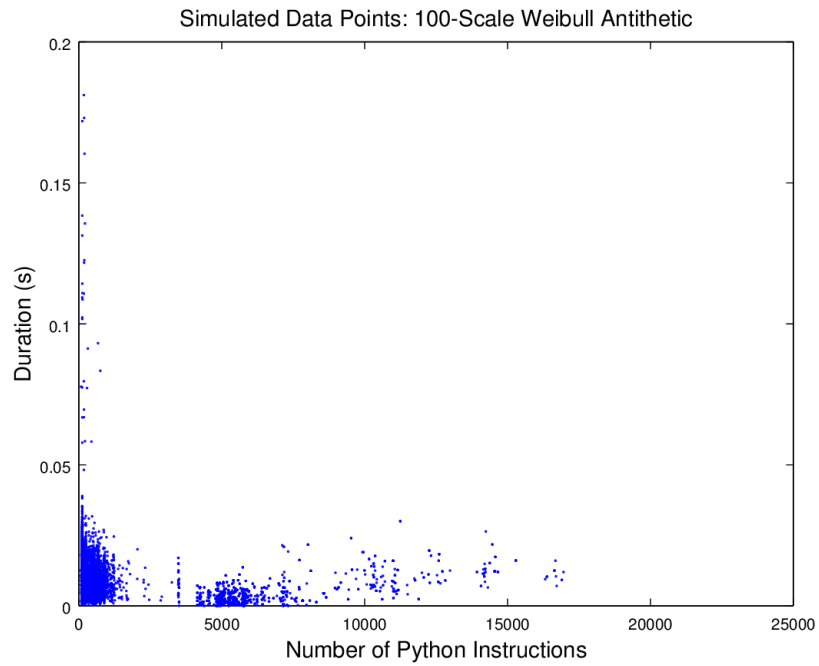


Figure A.12: Subset of simulated data for Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values are scaled by a factor of 100.

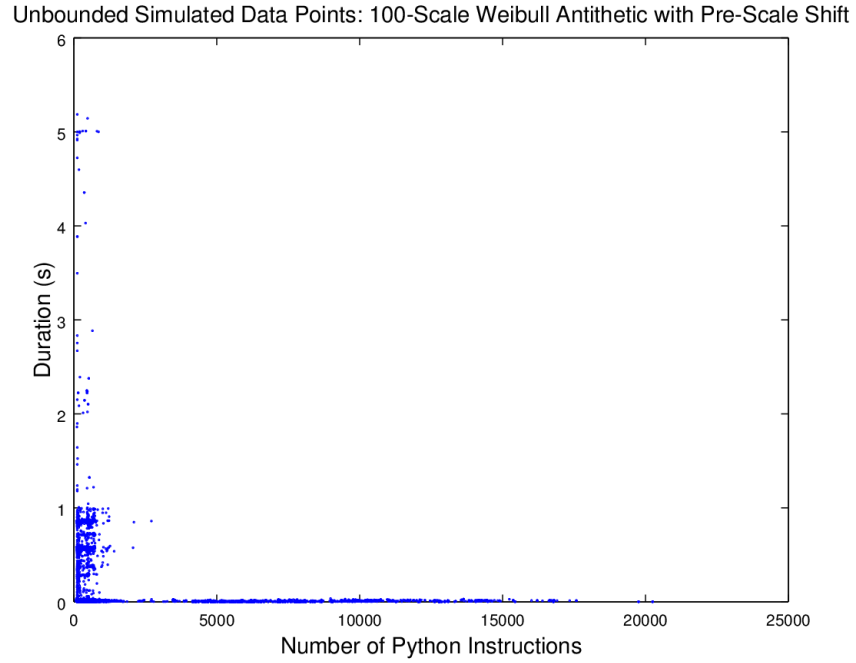


Figure A.13: Total simulated data for bound Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values (with a maximum of $5.22687e-4$) are shifted by $1.209e-5$, then scaled by a factor of 100.

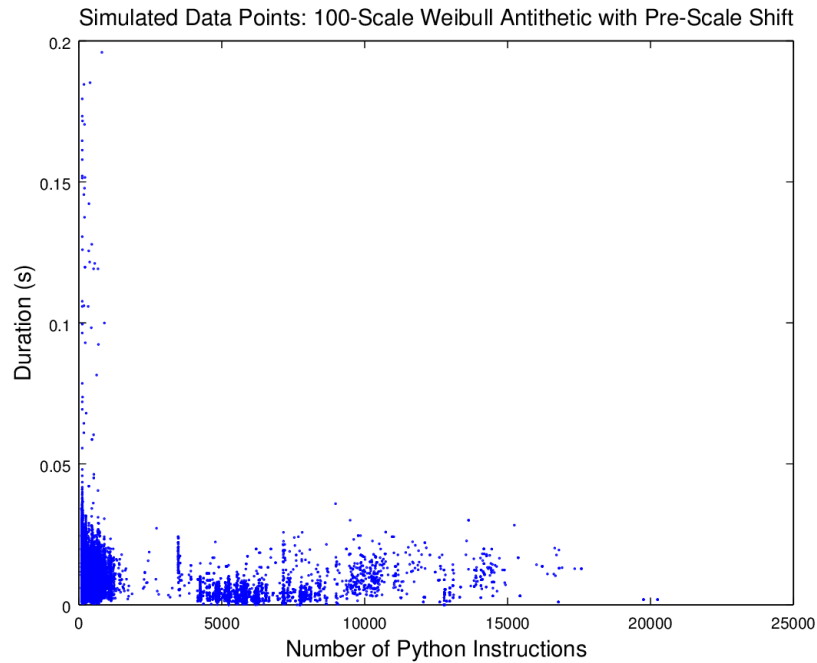


Figure A.14: Subset of simulated data for bound Weibull fitting with antithetic sampling with $scale=6.69475e-5$ and $shape=1.28053$ whose sampled values (with a maximum of $5.22687e-4$) are shifted by $1.209e-5$, then scaled by a factor of 100.

REFERENCES

- [1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, “Ethane: taking control of the enterprise,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007.
- [2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
- [3] B. Pfaff, B. Heller, D. Talayco, D. Erickson, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Pettit, K. Yap, M. Casado, M. Kobayashi, N. McKeown, P. Balland, R. Price, R. Sherwood, and Y. Yiakoumis, *Openflow switch specification, version 1.0.0 (wire protocol 0x01)*, Website, <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>, 2009.
- [4] O. N. Foundation, *Technical library - open networking foundation*, Website, <https://www.opennetworking.org/sdn-resources/technical-library>, 2015.
- [5] B. Pfaff, B. Lantz, B. Heller, C. Barker, C. Beckman, D. Cohn, D. Talayco, D. Erickson, D. McDysan, D. Ward, E. Crabbe, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Tonsing, J. Pettit, K. Yap, L. Poutievski, L. Vicisano, M. Casado, M. Takahashi, M. Kobayashi, N. Yadav, N. McKeown, N. d’Heureuse, P. Balland, R. Ramanathan, R. Price, R. Sherwood, S. Das, S. Gandham, T. Yabe, Y. Yiakoumis, and Z. Kis, *Openflow switch specification, version 1.3.0 (wire protocol 0x04)*, Website, <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf>, 2012.
- [6] *Ryu resources*, Website, <http://osrg.github.io/ryu/resources.html#documentation>, 2015.
- [7] *About POX — noxrepo*, Website, <http://www.noxrepo.org/pox/about-pox/>, 2015.
- [8] *Floodlight openflow controller - project floodlight*, Website, <http://www.projectfloodlight.org/floodlight/>, 2017.
- [9] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, “The design and implementation

- of open vswitch,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, 2015, pp. 117–130, ISBN: 978-1-931971-218.
- [10] D. Camara, H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage, “DCE: test the real code of your protocols and applications over simulated networks,” *Communications Magazine, IEEE*, vol. 52, no. 3, pp. 104–110, 2014.
 - [11] E. Bikov and P. Boyko, “Direct execution of OLSR MANET routing daemon in ns-3,” in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools ’11, Barcelona, Spain: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 454–461, ISBN: 978-1-936968-00-8.
 - [12] S. Kwon, K. Hasan, M. Lee, and S. Jeong, “Comparative analysis of real-time video performance in the CCN-based LTE networks,” in *Information and Communication Technology Convergence (ICTC), 2015 International Conference on*, 2015, pp. 509–511.
 - [13] H. Tazaki, E. Mancini, D. Camara, T. Turletti, and W. Dabbous, “MSWIM demo abstract: direct code execution: increase simulation realism using unmodified real implementations,” in *Proceedings of the 11th ACM International Symposium on Mobility Management and Wireless Access*, ser. MobiWac ’13, Barcelona, Spain: ACM, 2013, pp. 29–32, ISBN: 978-1-4503-2355-0.
 - [14] A. Roy, K. Yocum, and A. C. Snoeren, “Challenges in the emulation of large scale software defined networks,” in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, ser. APSys ’13, Singapore, Singapore: ACM, 2013, 10:1–10:6, ISBN: 978-1-4503-2316-1.
 - [15] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX, Monterey, California: ACM, 2010, 19:1–19:6, ISBN: 978-1-4503-0409-2.
 - [16] Y. J. Cheng, D. Huang, C. L. Lee, M. C. Lee, B. W. Chuang, M. C. Tsai, X. Huang, and C. H. Hsu, “Towards a detailed openflow emulator,” in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, 2015, pp. 127–132.
 - [17] P. Danielis, V. Altmann, J. Skodzik, E. B. Schweissguth, F. Golasowski, and D. Timmermann, “Emulation of sdn-supported automation networks,” in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, 2015, pp. 1–8.

- [18] J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield, "Efficient network-wide flow record generation," in *INFOCOM, 2011 Proceedings IEEE*, 2011, pp. 2363–2371.
- [19] M. Gupta, J. Sommers, and P. Barford, "Fast, accurate simulation for SDN prototyping," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: ACM, 2013, pp. 31–36, ISBN: 978-1-4503-2178-5.
- [20] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic sdn debugging," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, Sydney, Australia: ACM, 2014, pp. 227–234, ISBN: 978-1-4503-3279-8.
- [21] B. Hurd, *Openflow switch support – model library*, Website, <https://www.nsnam.org/docs/models/html/openflow-switch.html>, 2011.
- [22] P. Jurkiewicz, *Link modeling using ns-3*, Website, <https://github.com/mininet/mininet/wiki/Link-modeling-using-ns-3>, 2013.
- [23] M. Chan, C. Chen, J. Huang, T. Kuo, L. Yen, and C. Tseng, "Opennet: a simulator for software-defined wireless local area network," in *Wireless Communications and Networking Conference (WCNC), 2014 IEEE*, 2014, pp. 3332–3336.
- [24] D. Klein and M. Jarschel, "An openflow extension for the OMNeT++ INET framework," in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, ser. SimuTools '13, Cannes, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013, pp. 322–329, ISBN: 978-1-4503-2464-9.
- [25] S. Wang, C. Chou, and C. Yang, "Estinet openflow network simulator and emulator," *Communications Magazine, IEEE*, vol. 51, no. 9, pp. 110–117, 2013.
- [26] S. Y. Wang, "Comparison of sdn openflow network simulator and emulators: estinet vs. mininet," in *2014 IEEE Symposium on Computers and Communications (ISCC)*, 2014, pp. 1–6.
- [27] V. Antonenko and R. Smelyanskiy, "Global network modelling based on mininet approach," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: ACM, 2013, pp. 145–146, ISBN: 978-1-4503-2178-5.
- [28] A. Roy, M. Bari, M. Zhani, R. Ahmed, and R. Boutaba, "Design and management of dot: a distributed openflow testbed," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, 2014, pp. 1–9.

- [29] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003.
- [30] *Geni-lib 0.9.2 documentation*, Website, <http://geni-lib.readthedocs.io/en/latest/>, 2014.
- [31] B. P. Swenson and G. F. Riley, "Simulating large topologies in ns-3 using brite and cuda driven global routing," in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, ser. SimuTools '13, Cannes, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013, pp. 159–166, ISBN: 978-1-4503-2464-9.
- [32] G. F. Riley, M. H. Ammar, and E. W. Zegura, "Efficient routing using nix-vectors," in *High Performance Switching and Routing, 2001 IEEE Workshop on*, 2001, pp. 390–395.
- [33] G. F. Riley, M. H. Ammar, and R. Fujimoto, "Stateless routing in network simulations," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, 2000, pp. 524–531.
- [34] G. F. Riley and D. Reddy, "Simulating realistic packet routing without routing protocols," in *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, 2005, pp. 151–158.
- [35] Z. Hao, X. Yun, and H. Zhang, "An efficient routing mechanism in network simulation," in *20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*, 2006, pp. 150–157.
- [36] Y. J. Lee and G. F. Riley, "Dynamic nix-vector routing for mobile ad hoc networks," in *IEEE Wireless Communications and Networking Conference, 2005*, vol. 4, 2005, 1995–2001 Vol. 4.
- [37] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948–960, 1972.
- [38] W. Wu and P. Demar, "A gpu-accelerated network traffic monitoring and analysis system," in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, 2013, pp. 77–78.
- [39] Y. Li, D. Zhang, A. X. Liu, and J. Zheng, "Gamt: a fast and scalable ip lookup engine for gpu-based software routers," in *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, 2013, pp. 1–12.

- [40] A. Nottingham and B. Irwin, “A high-level architecture for efficient packet trace analysis on gpu co-processors,” in *2013 Information Security for South Africa*, 2013, pp. 1–8.
- [41] ———, “Parallel packet classification using gpu co-processors,” in *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, ser. SAICSIT ’10, Bela Bela, South Africa: ACM, 2010, pp. 231–241, ISBN: 978-1-60558-950-3.
- [42] N. P. Tran, M. Lee, S. Hong, and J. Choi, “High throughput parallel implementation of aho-corasick algorithm on a gpu,” in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, 2013, pp. 1807–1816.
- [43] C. H. Lin, S. Y. Tsai, C. H. Liu, S. C. Chang, and J. M. Shyu, “Accelerating string matching using multi-threaded algorithm on gpu,” in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, 2010, pp. 1–5.
- [44] N. F. Huang, H. W. Hung, S. H. Lai, Y. M. Chu, and W. Y. Tsai, “A gpu-based multiple-pattern matching algorithm for network intrusion detection systems,” in *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, 2008, pp. 62–67.
- [45] F. Shao, Z. Chang, and Y. Zhang, “Aes encryption algorithm based on the high performance computing of gpu,” in *Communication Software and Networks, 2010. ICCSN ’10. Second International Conference on*, 2010, pp. 588–590.
- [46] C.-L. Hsieh and N. Weng, “Many-field packet classification for software-defined networking switches,” in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS ’16, Santa Clara, California, USA: ACM, 2016, pp. 13–24, ISBN: 978-1-4503-4183-7.
- [47] K. Qiu, Z. Chen, Y. Chen, J. Zhao, and X. Wang, “Gflow: towards gpu-based high-performance table matching in openflow switches,” in *2015 International Conference on Information Networking (ICOIN)*, 2015, pp. 283–288.
- [48] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, “Optimizing many-field packet classification on fpga, multi-core general purpose processor, and gpu,” in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, 2015, pp. 87–98.
- [49] M. Varvello, R. Laufer, F. Zhang, and T. V. Lakshman, “Multilayer packet classification with graphics processing units,” *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–1, 2015.

- [50] W. Sun and R. Ricci, “Fast and flexible: parallel packet processing with gpus and click,” in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS ’13, San Jose, California, USA: IEEE Press, 2013, pp. 25–36, ISBN: 978-1-4799-1640-5.
- [51] T. Suzuki, S. Y. Kim, J. i. Kani, K. I. Suzuki, A. Otaka, and T. Hanawa, “Parallelization of cipher algorithm on cpu/gpu for real-time software-defined access network,” in *2015 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, 2015, pp. 484–487.
- [52] E. G. Renart, E. Z. Zhang, and B. Nath, “Towards a gpu sdn controller,” in *Networked Systems (NetSys), 2015 International Conference and Workshops on*, 2015, pp. 1–5.
- [53] J. Ivey and G. Riley, “Analysis of programming language overhead in dce,” in *Proceedings of the Workshop on Ns-3*, ser. WNS3 ’16, Seattle, WA, USA: ACM, 2016, pp. 41–48, ISBN: 978-1-4503-4216-2.
- [54] J. Ivey, G. Riley, B. Swenson, and M. Loper, “Designing and enabling simulation of real-world gpu network applications with ns-3 and dce,” in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016, pp. 445–450.
- [55] J. Ivey, H. Yang, C. Zhang, and G. Riley, “Comparing a scalable sdn simulation framework built on ns-3 and dce with existing sdn simulators and emulators,” in *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS ’16, Banff, Alberta, Canada: ACM, 2016, pp. 153–164, ISBN: 978-1-4503-3742-7.
- [56] —, “Comparing an sdn simulation framework in ns-3 and dce with existing sdn simulators and emulators,” *Submitted to ACM Transactions on Modeling and Computer Simulation (TOMACS)*,
- [57] L. J. Chaves, I. C. Garcia, and E. R. M. Madeira, “Ofswitch13: enhancing ns-3 with openflow 1.3 support,” in *Proceedings of the Workshop on ns-3*, ACM, 2016, pp. 33–40.
- [58] E. L. Fernandes and C. E. Rothenberg, “Openflow 1.3 software switch,” *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuidos SBRC*, pgs, pp. 1021–1028, 2014.
- [59] *Openflow 1.3 software switch by cpqd*, Website, <http://cpqd.github.io/ofsoftswitch13/>.

- [60] *The netbee library [netbee]*, Website, <http://www.nbee.org/doku.php>, 2012.
- [61] D. Camara, *Bake: main page view*, Website, <http://planete.inria.fr/software/bake/index.html>, 2015.
- [62] B. Pfaff, B. Lantz, B. Heller, C. Barker, D. Cohn, D. Talayco, D. Erickson, E. Crabbe, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Pettit, K. Yap, L. Poutievski, M. Casado, M. Takahashi, M. Kobayashi, N. McKeown, P. Balland, R. Ramanathan, R. Price, R. Sherwood, S. Das, T. Yabe, Y. Yiakoumis, and Z. Kis, *Openflow switch specification, version 1.0.0 (wire protocol 0x01)*, Website, <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf>, 2011.
- [63] R. W. Floyd, “Algorithm 97: shortest path,” *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.
- [64] *Pycuda*, Website, <https://mathematician.de/software/pycuda/>, 2017.
- [65] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, “Geni: a federated testbed for innovative network experiments,” *Computer Networks*, vol. 61, no. 0, pp. 5–23, 2014, Special issue on Future Internet Testbeds Part I.
- [66] D. Nicol, *Ssfnet gallery*, Website, <http://www.ssfnet.org/Exchange/gallery/baseline/index.html>.
- [67] J. Pelkey and G. Riley, “Distributed simulation with mpi in ns-3,” in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools ’11, Barcelona, Spain: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 410–414, ISBN: 978-1-936968-00-8.
- [68] B. P. Swenson, J. S. Ivey, and G. F. Riley, “Performance of conservative synchronization methods for complex interconnected campus networks in ns-3,” in *Proceedings of the 2014 Winter Simulation Conference*, ser. WSC ’14, Savannah, Georgia: IEEE Press, 2014, pp. 3096–3106.
- [69] *Numpy – numpy*, Website, <http://www.numpy.org/>, 2016.
- [70] *Random variables – manual*, Website, <https://www.nsnam.org/docs/manual/html/random-variables.html>, 2017.

VITA

Jared S. Ivey is a technical lead engineer for software development and maintenance of the Joint STARS E-8C platform at Robins Air Force Base, GA. He received his B.S. in Biomedical Engineering from the Georgia Institute of Technology in May 2009 and his M.S.E. in Software Engineering from Mercer University in May 2012. He is currently pursuing a Ph.D. in Electrical and Computer Engineering under the supervision of Dr. George F. Riley at the Georgia Institute of Technology. His research interests focus on providing scalable and reliable solutions for network simulation of software-defined networks. Outside of work, he regularly engages in long-distance running, achieving a 17:48 time in the 5K and 1:25:47 in the half-marathon as personal records. He also enjoys listening to podcasts on design, history, and running. He married Sloane Oldham Ivey in March 2011, and the two of them enjoy traveling and attending Georgia Tech football games. In April 2017, they welcomed the birth of their first son, Knowles, as well as all of the ups and downs of the roller coaster that is parenthood.